PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

A MICROPROGRAMMED INTERPRETER FOR THE PERSONAL SEQUENTIAL INFERENCE MACHINE

Minoru Yokota, Akira Yamamoto
Kazuo Taki, Hiroshi Nishikawa, Shunichi Uchida
Institute for New Generation Computer Technology

Katsuto Nakajima
Mitsubishi Electric Co.,Ltd

Masaki Mitusi
Oki Electric Industry Co.,Ltd

ABSTRACT

As a part of Fifth Generation Computer Systems (FGCS) project, a personal sequential inference machine, PSI, has been designed for software development. PSI has a logic based machine language, FGCS Kernel Language Version 0 (KL0), which is similar to DEC-10 Prolog. One of the main features of PSI is that this high-level machine language is executed directly in firmware, a microprogrammed interpreter.

The microprogrammed interpreter is similar to the existing Prolog system and uses "structure sharing" for representing structured data. However, its unification mechanism is based on "argument copying", and it also provides powerful new control structures, such as "remote cut" and "bind hook", for practical use. The several primitive functions required to build the operating system on PSI, such as interrupt handling and memory allocation, are also implemented as a part of the microprogrammed interpreter. This interpreter fully takes advantages of microprogramming technique and the features of specialized hardware of PSI.

1 INTRODUCTION

A Personal Sequential Inference Machine, PSI, has been designed as a software development tool (Yokota 83), (Uchida 83), (Nishikawa 83). Its machine instruction set is designed to be of the same level as the FGCS Kernel Language Version 0, called KL0 (Chikayama 83). KL0 is a logic programming language based on Prolog and includes most prolog functions, except for such data base handling as "assert" and "retract". However, KL0 is intended to be used primarily for writing the operating system, as well as for developing application programs. For this reason, specific functions related to execution control and low-level system description have been introduced in KL0.

One of the critical design decisions for any new machine architecture is the determination of the level of its machine instruction set. If the machine instructions are designed to be low-level, each instruction has simple functions and it can be executed quickly. However, the actual execution speed depends on whether the compiler can generate optimized object code. If machine instructions are designed of the same high level as source statements, hardware/firmware takes responsibility for fast program execution.

Since conventional machines have relatively low-level instructions, existing Prolog systems have implemented by the former approach. Recently, more sophisticated machine architecture based on the former approach has been proposed (Tick 84). However, we considered that high-level language machine approach is better for implementing PSI for the following reasons:

(1) Dynamic features in logic programming

In logic programming system, variables are usually type-less and their data types are determined at execution time. The backtracking mechanism provides non-deterministic feature in program execution. The compiler cannot generates the deterministic code for such dynamic execution, and produces the interpretive code for them. To speed program execution using compiler, the user specifies explicitly the behavior of the program and restricts its generality. However, these dynamic features produce several benefits to users. The architectural support for them is desirable.

(2) Simple interpretation mechanism

The basic interpretation mechanism of Prolog is quite simple. If we close-up its unification process, for example, it may be shown that fetching the values of caller and callee arguments and comparing them are the most dominant operations. Several data types must be manipulated in the unification process, and the number of instruction steps required for each type is usually small. Even for structured data, complexity results from its nested structure and a similar small number of steps is usually sufficient for each structure element. Microprogramming techniques, such as powerful multi-way conditional branching, are suitable for implementing this unification procedure. For fast data fetching, the interface between CPU and the memory unit has been designed as simple as possible in PSI.

(3) Small interpreter kernel

Since the interpretation mechanism is quite simple, small number of steps are required for the interpreter kernel. This means that even if the kernel is implemented completely in microcode it is manageable and hand-optimizable. We estimated the size of the kernel at approximately 1K steps.

(4) Flexible microprogramming

KL0 was originally designed at ICOT and its language specification will be improved in future to reflect our experience with it. The flexibility of microprogram implementation is desirable for this reason. In addition, it enabled us to embed debugging and evaluation facilities in the interpreter with little additional overhead time.

In keeping with the above considerations, the interpreter system for PSI has been implemented completely in microcode and is called "PSI *microinterpreter* ". The following section provides an overview of the *microinterpreter* and its basic interpretation mechanism. Section 3 describes the main features of its kernel control, Section 4 describes special event-driven control, and Section 5 describes the relationship to the operating system.

2 OVERVIEW OF THE MICROINTERPRETER

Since the machine language KL0 is similar to the logic programming language Prolog, the functions performed in the *microinterpreter* are basically the same as those of existing Prolog interpreters.

Unification and backtracking are performed at the microprogram level.

Some primitive functions and frequently used functions are incorporated as built-in predicates, and are executed directly with no definition.

Since KL0 is PSI's machine language, it must include sufficient functions to implement PSI operating system, which works as a stand-alone system. These functions usually require low-level and hardware dependent operations. In result, the *microinterpreter* must support both pure logic programming functions for user's application system and low-level system functions for its operating system in a uniform manner. Some functions are incorporated as built-in predicates, and others are performed by system support functions of the *microinterpreter* .

2.1 Organization of MicroInterpreter

The *microinterpreter* consists of the following three parts, as shown in Figure 1.

- the kernel
- built-in predicates
- operating system interface

(1) Kernel

The kernel implements pure logic programming functions such as call/return execution control and unification. Since the unification process is almost independent of execution control, it is implemented as a separate microprogram routine. Every user-defined predicate is interpreted within this kernel.

(2) Built-in predicates

The *microinterpreter* incorporates many built-in predicates to support primitive operations or frequently used operations. These predicates are implemented directly in microcode. The typical usage of them are listed below:

- Primitive operations

  ex. atom(X), unbound(X)

- Rewriting operations with side effects

  ex. set-vector-element(Vector,Position,Element)

- Frequently used and determinate operations

  ex. add(X,Y,Z), increment(X,Y)

- Hardware resource manipulation

  ex. set-register(Register,Tag,Value)

- Operating system support

  ex. change-process(ProcessNumber)

- Special execution control

  ex. cut, fail, bind-hook(X,handler(X))

The evaluation process for each built-in predicate is invoked from the kernel. Namely, when such a predicate appears in a clause, the kernel executes indirect jump to the object microprogram routine according to the operation code. This process takes advantage of the special dispatching hardware. The routine evaluates the predicate's arguments, executes the designated operation, unifies its results to the given arguments, and finally returns control to the kernel. Some built-in predicates overwrite the arguments by their results instead of unification.

(3) Operating system interface

Several built-in predicates are provided for operating system support. the primitive system operations are performed implicitly as operating system interface functions. These include process switching, physical page allocation, interrupt handling, and exception handling.

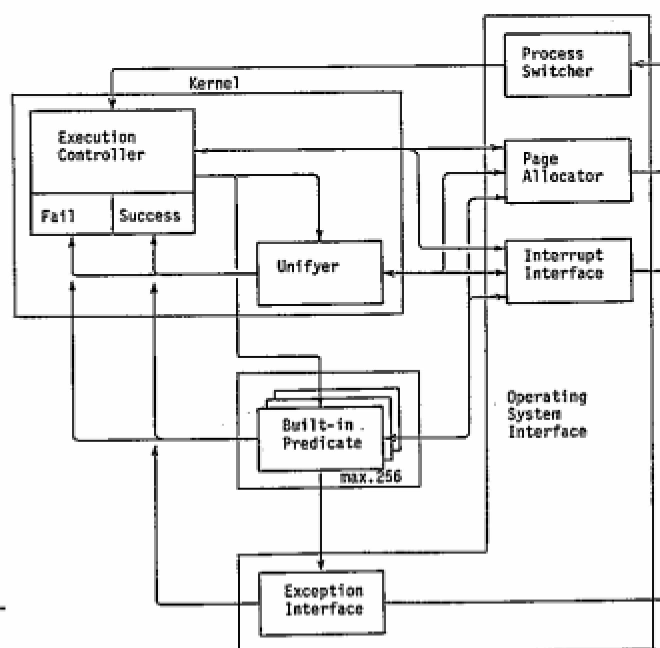

Figure 1. The Structure of the MicroInterpreter

For KL0 interpretation, several stack and heap areas are required. Their allocation is effectively controlled at the microprogram level. Whenever additional memory space is required, the page allocator is called. This routine examines the memory environment and, if necessary, allocates a new page.

Interrupts are checked in the kernel or in particular built-in predicates. When an interrupt is detected at one of these check points, the interrupt interface is called. This interface routine examines the interrupt source, determines the corresponding handler process number, and finally, exchanges the current process to the handler process.

Exceptions are treated similarly to interrupts, except that no process switching occurs. The exception interface routine examines the exception source and invokes the corresponding handler predicate.

The process switcher saves the whole environment of the current process, and loads the new environment of the called process. Since the saved environment includes the microprogram counter, the interrupted process can continue its execution from the latest interrupt point of the *microinterpreter*. The process switching is also caused by the detection of unusual conditions, such as no physical page or no exception handler. This mechanism is defined as a "trap".

2.2 Basic Execution Mechanism

The interpretation mechanism of the kernel is based on DEC-10 Prolog (Warren 77). However, several useful data types, such as vectors and strings, have been introduced, and powerful execution control mechanism has been provided. Since the control stack is separated from the local stack to achieve this sophisticated execution control, as a result, the *microinterpreter* uses four independent stacks: Local, Global, Control, and Trail.

Figure 2 illustrates internal object form corresponding to a KL0 sample program. A set of the clauses which have the same head predicate is called a "procedure". It is packed into a data block and stored in the heap area. Each predicate argument is represented as an ordinary data type. If it appears in a head predicate, it is interpreted as a unification instruction and its tag is regarded as an operation code. There is no explicit instruction for unification and backtracking in the internal form.

The goal is translated into a code type data pointing to the callee procedure code, and the required arguments follow it. The invocation link from a caller goal to the callee procedure is established by the compiler, and using this link, the *microinterpreter* gets the information about the first clause of the callee procedure. Then, the *microinterpreter* creates callee's variable cells and the arguments of caller goal are unified with those of the callee head.

If unification is completed successfully, the current environment is pushed on the top of the control stack, and the callee's body goal (q(X) in this example) is evaluated next in the same manner described above. If the invoked predicate is a unit clause, control returns to the caller and the next body goal of the caller is evaluated. On the other hand, if unification fails, the *microinterpreter* gets the alternative clause and tries it.

During the above operations, several base registers are maintained as the execution environment. These are the base address of the callee variable cell area, caller and callee argument addresses, and so on.

The use of each stack is similar to that in other Prolog systems. A local stack is used to store the values of local variables. Similarly, a global stack is used for global variables, which represent variables in structured data. However, it also stores molecules and other control information, such as an exception control block. A control stack is used for saving the execution environment which is required at returning to the caller or at backtracking to the alternatives. A trail stack is used to reset variable bindings during backtracking. For this aim, cell addresses of the bound variables are saved in the trail stack. To reset the control environment, the old values are also saved in the trail stack along with their memory addresses.

The PSI hardware has been designed to efficiently interpret KL0 and is equipped with specialized support mechanisms for checking tags, conditional jump by tag, stacking variable cells, and so on (Taki 84).

3 FEATURES OF THE KERNEL

3.1 Argument Copying with Frame Buffer

In the unification process, the values of both caller and callee arguments are dereferenced and compared with each other. The *microinterpreter* divides this process into two phases. One is fetching and evaluating the caller arguments and setting their values on the top of the local

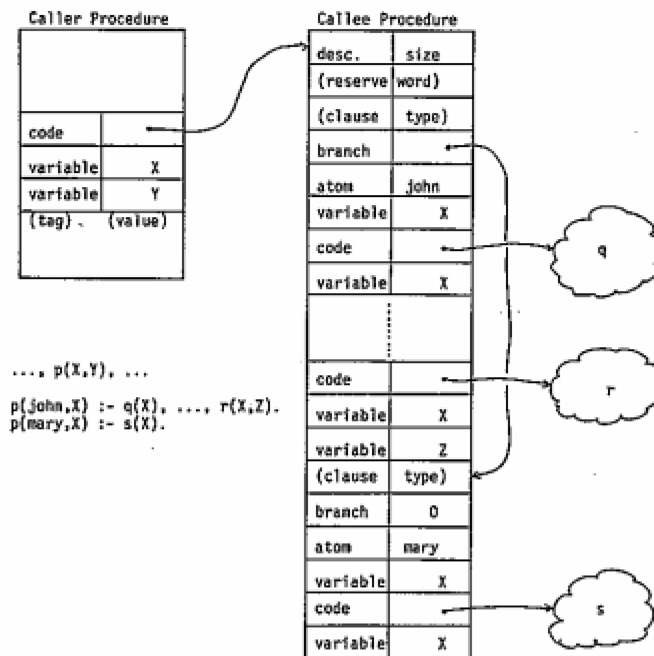

Figure 2. An Example of Internal Object Form

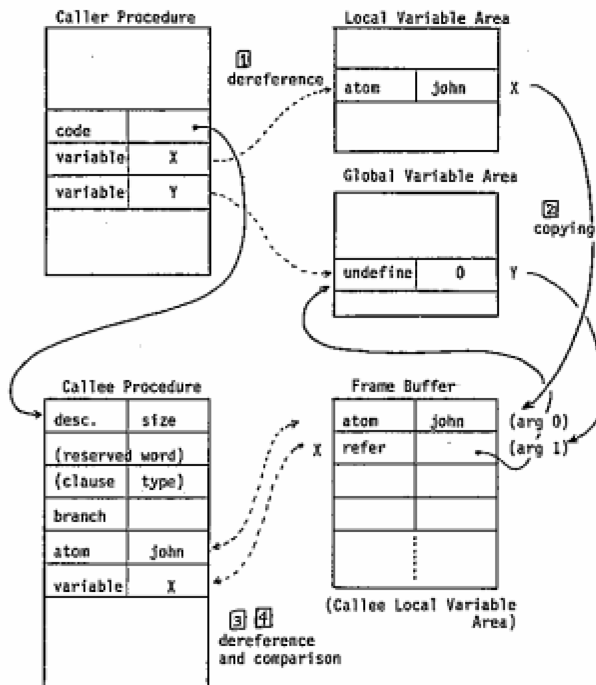

Figure 3. Argument Copying

stack. The other is fetching and evaluating the callee arguments and comparing their values with the copied caller argument values. Figure 3 shows this process. In this way, the caller arguments are copied to the callee environment before unification. The same unification mechanism has been used as in Tail Recursion Optimization for the DEC-10 Prolog system (Warren 80). However, the *microinterpreter* employs this method not only in tail recursion, but in every goal call. We call this unification method "argument copying".

Argument copying has the following advantages.

(1) If a goal is the last in a caller clause and that clause will complete deterministically, its environment need not be saved for backtracking. Since required caller argument values are copied into the callee environment by argument copying, the disused caller environments can be discarded before unification. Especially in recursive predicate calls, the same memory space is used for both caller and callee (Tail Recursion Optimization).

(2) When unification fails and the *microinterpreter* must backtrack to alternative clauses of the same procedure, the copied caller arguments can be used again. It is not necessary to re-evaluate caller arguments. This feature speeds the backtracking process.

(3) Since the callee argument evaluation process is separate from that of the caller, unification mechanism is simplified.

The one disadvantage of argument copying is the increases in execution time and memory space by the copying process itself.

To decrease this copying overhead, the copying area is overlapped with the callee local variable area. In this implementation, no operation is required for callee's unbound variables (callee's X in Figure 3), because the corresponding caller argument values have already been set in those variable cells by the copying operation. To easily detect such unbound variables, a tag is provided for variables which appears first within a clause. This tag guarantees that the value of a variable is unbound.

In addition to this optimization, PSI has a special hardware buffer, called a "frame buffer", which holds the top 31 words of the local stack. Since the caller arguments are copied to this buffer, access to them is very fast. Furthermore, PSI has two frame buffers. If the small loop operation is written by the tail recursive predicate call, their execution can be performed within this hardware buffer by switching the frame buffer alternatively.

3.2 Inner Clause OR

Alternative clauses in Prolog are considered to be connected by the OR function. However, OR connection can be defined in a clause, as shown below.

(X,Y) :- ..., (r(X) ; s(Y)), ...

We distinguish this type of OR connection as "inner clause OR". One of the advantages of inner clause ORs is that argument passing can be eliminated. For example, the above example could be written using alternative clauses, as follows:

p(X,Y) :- ..., temp(X,Y), ...

temp(X,—) :- r(X).
temp(—,Y) :- s(Y).

However, the unification process for calling temp(X,Y) is obviously redundant, as compared with the first example. If callee clauses have many and complex head arguments, overhead becomes a significant factor.

The main reason for separating the control stack from the local stack is the implementation of this inner clause OR. The goals within the inner clause OR share the variable cells with the parent clause which involves this inner

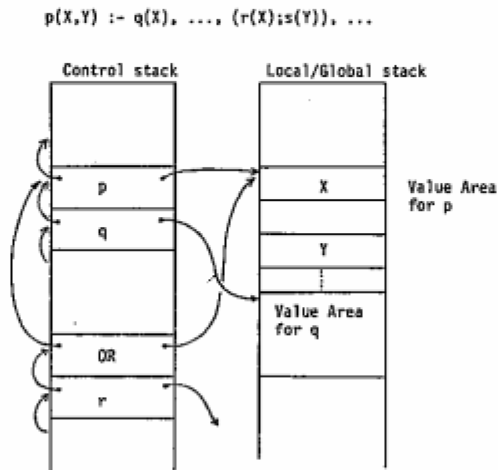

Figure 4. Environment of the Inner Clause OR

clause OR, as shown in Figure 4. However, its execution level is different from that of the parent clause.

3.3 Execution Control by Invocation Level

Prolog program execution can be controlled by the built-in predicates, "cut" and "fail". However, these are not always sufficient to cover many practical requirements. The execution control of KL0 has been improved. One of the extended control facilities is the "remote cut" operation. The function of the remote cut is to cut the alternative clauses up to an arbitrary point. Initially, we designed this mechanism so that the programmer could label the desired position, then cut up to this label.

p(X,Y) :- ..., mark(12), q(Y).

r(X) :- ..., cut-up-to(12), fail.

However, this strategy caused complex label management problems. One of these involved dealing with label information for other remote cut operations that would discard the label. Finally, we solved this problem by introducing the execution level. Specifically, each execution environment is distinguished by the invocation level.

Using the level number, the user can cut the disused alternative clauses to the any level, relatively (relative-cut(Level)), or absolutely (absolute-cut(Level)). For this, the *microinterpreter* has a level counter, which is incremented whenever a goal is called. This execution level is also saved in the control stack as an execution environment, and is used to reset the level counter to the caller's execution level.

By combining the remote cut with "fail", the fail-throw mechanism of "Catch and Throw" can be implemented. Since it is expected that the fail-throw mechanism will be used frequently, the built-in predicates, "absolute-cut-and-fail" and "relative-cut-and-fail", are provided. For success-throw mechanism, the built-in predicate, "succeed", is provided. This predicate also uses the level number to return to an arbitrary execution level. In the next example, the predicate r(Y) will be invoked after execution of succeed(4).

p(X) :- ..., q(X), r(Y), ... (level 4)

q(X) :- ..., s(X), ... (level 5)

s(X) :- ..., succeed(4). (level 6)

This execution level is also useful for debugging KL0 programs. For example, it is possible to eliminate unnecessary trace output by using the execution level.

3.4 Predicate Call Mechanism

The link from a caller predicate to the callee procedure is usually established at compile time. Clause invocation is performed using this link as described in Section 2. However, the *microinterpreter* supports more flexible and powerful predicate call mechanisms, as described below.

(1) Indirect Call

Indirect call is a predicate call that invokes the object procedure through indirection pointers. This type of call is useful for cases in which the procedure to be called is dynamically determined, and thus, its location is not known at compile time. For example, if a user program requires a operating system routine, however, its predicate address cannot be determined at compile time. The compiler generates an invocation link to the indirect word in this case, and the operating system completes this link at loading time by filling the required procedure address into the indirection word. This indirect call mechanism is also useful for debugging, because the object procedure can be dynamically exchanged from the original one to the debug-support procedure by exchanging this indirection word.

(2) Dynamic Call

The built-in predicate "apply(Code,Arguments)" allows the user to invoke an arbitrary procedure with arbitrary arguments at runtime. The micro routine checks whether the fist argument of "apply" is a code, and checks whether the second argument is a vector having the correct number of arguments for that code. If all the input conditions are satisfied, the designated procedure is invoked in the same way as an ordinary predicate call. For example, if X is bound to the code "p(X,Y) :- q(X).", and Y is bound to the two element vector "a,Z",

..., apply(X,Y), ...

acts as if it were written

..., p(a,Z), ...

(3) Clause Indexing

Clause indexing is a predicate call in which the object predicate (not procedure) is selected according to the value of one of the caller arguments (Warren 77). This type of predicate call is very effective if the callee procedure has many alternative clauses, as in the case of a data

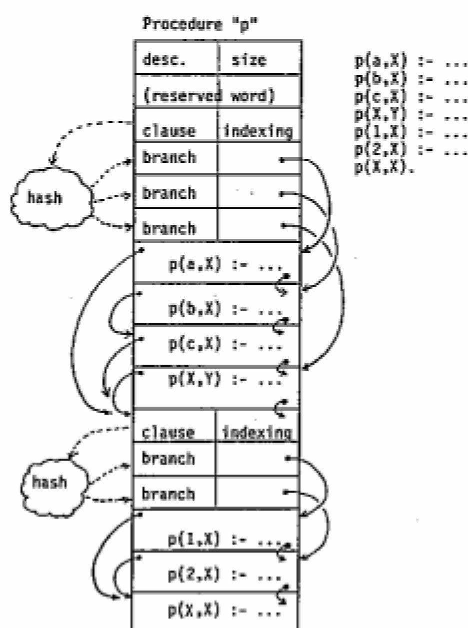

Figure 5. The Internal Object Form for Clause Indexing

base access. Figure 5 shows the internal object form for clause indexing. If the index argument remains undefined, indexing does not apply and all alternative clauses must be tried. Therefore, each clause should have two alternative chains: one as an ordinary chain and the other for clause indexing. The *microinterpreter* has a state flag, which indicates the indexing mode. When backtrack occurs, the next alternative clause is fetched along with one of these two chains, depending on the state flag. In DEC-10 Prolog, the only first argument can be used as an index. However in PSI, the index argument is determined by the compiler, and its argument number is set in the internal object form.

4 SPECIAL EVENT-DRIVEN CONTROL

The microprogrammed implementation enables many dynamic features with little increase in overhead. Taking full advantage of this, the *microinterpreter* provides a dynamic predicate call mechanism, which is triggered on special events. A procedure can be waked up by binding during unification, by backtracking, or by detecting illegal conditions during evaluation of a built-in predicate.

4.1 Call on Binding (Bind-Hook)

The unification-driven predicate call mechanism, called "bind-hook", is introduced to know when a variable is bound and what is bound to it. This can be effectively used for sophisticated programming and debugging. For this implementation, the data type "hooked variable" is introduced. This data type indicates for an unbound variable, and the procedure address waiting for its binding. Whenever a value is set to some undefined variable during unification, the *microinterpreter* examines whether the destination cell has a hooked variable. If it does, the waiting procedure is invoked after unification.

At first, the variable and the code to be called are defined by the built-in predicate "bind-hook(X,Code)".

..., bind-hook(X, (p(X,Y),q(Y), ...)), ...

The *microinterpreter* checks whether the given variable is still unbound, then sets the code address in its cell as a hooked variable. In the above example, a memory address of the code ":- p(X,Y),q(Y), ..." is set into X.

When the "hooked variable" is found during unification and some value is going to be set to it, the waiting procedure address is saved before binding. After the unification process has completed, the required procedure is invoked.

..., bind-hook(X, (p(X,Y),q(Y))), ..., r(X,Y).

r(X,Y) :- ..., s(X,Z), ...

s(a,X) :- t(X).

The last clause will be executed just as below.

s(a,X) :- p(a,Y), q(Y), t(X).

If more than one "hooked variable" are found in an unification process, after it has been completed, the several procedures are invoked at the same time. These are executed like as being connected by AND function.

4.2 Call on Backtracking (On-Backtrack)

When a predicate fails, the remaining alternative clauses are evaluated(backtracking). If some alternatives become disused, they can be discarded by the built-in predicate "cut". However, it is difficult to partially discard the alternatives. Furthermore, there is no way to protect the necessary alternatives from deletion by the careless use of "cut".

The built-in predicate "on-backtrack(Code)" is provided to define the alternative clauses that are not to be affected by "cut". The following two clauses have the same effect and leave an alternative. The difference between them is that the alternative clause defined by "on-backtrack" cannot be eliminated by "cut".

..., on-backtrack((p(X),q(Y))), ...

..., (true ; p(X),q(Y)), ...

To implement this mechanism, the *microinterpreter* saves the given code address in the trail stack whose entry has the special tag, "on-backtrack-call". All variables used in the alternative clause must be compiled into global variables, which are not affected by "cut".

When backtracking occurs, the *microinterpreter* looks up the trail stack entries for the undo operation. If "on-backtrack-call" is found during this process, the *microinterpreter* suspends the backtrack process and invokes the specified code. In order to continue the backtrack, the invoked code should terminate fail. Therefore, the compiler automatically appends the built-in predicate "fail" to the end of the code.

4.3 Call on Exception (Exception-hook)

The built-in predicates are designed for efficient execution of primitive and frequently used operations. From this point of view, each built-in predicate has limited input conditions. The built-in predicate "add", for example, must have numeric data as its first and second arguments. If these input conditions are not satisfied, the *microinterpreter* raises exceptions and invokes a corresponding handler predicate, as bellow.

..., add(X,Y,Z), p(Z), ...

..., add(X,Y,Z), handler, p(Z), ...

There are 32 system-defined exceptions, and the code addresses of their corresponding exception handlers are located in the top 32 words of the global stack.

In addition, the user can define exceptions using the built-in predicate "exception-hook(Exception,Code)",

..., exception-hook(overflow, handler), ...

handler(X1,X2,...,X8) :- check(X3,X4), ...

and raise the exception at any time using the built-in predicate "raise(Exception,Arguments)".

..., raise(overflow,{1,2,...,8}), p(X), ...

In this case, the user-defined exception handler is waked up as follows.

handler(1,2,...,8) :- check(3,4), ...

5 RELATIONSHIP TO THE OPERATING SYSTEM

PSI has its own operating system based on logic programming. Several low-level built-in predicates are provided for access to PSI hardware resources, such as registers, WCS, main memory and I/O devices. Entire operating system can be written using only these predicates, as in conventional computer systems. However, this approach is obviously inefficient in the case of PSI. The load/store built-in predicate executes slowly because its operand data is fetched through the dereference operation. On the other hand, it is not practical to implement entire operating system in firmware. Designing the interface between the operating system and the hardware system is similar to designing the machine instruction level.

The operating system functions to be considered here can be divided into two groups. Memory allocation and interrupt handling are the examples of the first group and these are closely related to KL0 interpretation. For example, memory allocation for stacks and heaps depends on the usage of such area for interpretation. The operations required for interrupt handling depends on the interrupt check timing during interpretation. These functions are performed by the operating system interface routines described in Section 2.

The functions of the second group are independent of the KL0 interpretation mechanism. They are implemented as built-in predicates. A design decision had to be made as to which functions would be merged and implemented as a built-in predicate. Taken into consideration were efficiency of the system program execution, descriptiveness for non-logical low-level system operations, and clearness of the interface between firmware and software.

To decrease operating system overhead, frequently-used functions should be implemented in microcode. Process switching and garbage collection are examples of this type.

The load register and input/output functions have side effects, and these operations cannot be undone. Therefore, they should be implemented as built-in predicates as in the conventional machine instructions Load/Store or Input/Output.

If some of the operating system functions are implemented at the microprogram level, and if others that are closely related are implemented at the software level, the system control table may be referred from both firmware and software modules. To avoid cross-talk between firmware and software modules, and to keep clean the interface between them, related functions should be implemented either software or firmware, but not both. As a result of this design strategy, software and firmware can be modified independently.

5.1 Memory Management

PSI hardware has a large logical memory space of 4G words, divided into 256 independent segments, called "areas". Each area is managed in units of pages of 1K words. The area is specified by the following factors:

- Current area top address
- Current area size
- Corresponding page map base address
- User-defined area size limit
- Necessity of garbage collection

The above five factors form a block, Area Control Block (ACB). However, they are stored in different physical locations. For example, the current area top address is stored in the register file, and each page map base address is stored in the high-speed memory used for address translation. To manipulate them independently of the physical implementation, built-in predicates, such as "set-page-map-base(Area,NewBase)", are provided. Each factor of an area control block can be accessed using these built-in predicates. However, some predicates perform more complex functions. For example, if the user designates the smaller size than the current area size in "set-area-size(Area,NewSize)", the *microinterpreter* shrinks the size of the area, and returns the disused physical pages to the free physical page list.

As described in Section 2, dynamic memory allocation is performed as one of the *microinterpreter* functions. Since the memory space is allocated in a unit of pages, the page allocator is called by the occurrence of page boundary crossing. To effectively detect this situation, PSI hardware has a special carry flag, which is set by the page boundary overflow during address calculation.

At first, the page allocator checks whether the next page has already been allocated. If it does, there is noting to do. If the next page does not exit, the following conditions are examined.

- whether the new area size is within the user-defined

limit.

- whether the page map for address translation can be expanded.
- whether free physical pages are remaining.

If the above all conditions are satisfied, a free physical page is fetched from the free physical page list, and allocated to the area. If not, the page allocator raises a trap.

5.2 Process Management

To execute a KL0 program, the *microinterpreter* maintains several base addresses. Some examples are the top addresses of the four stacks and the base addresses of the caller and callee variable cell areas. The key addresses that is indispensable for backtracking and returning control are saved in the control stack. The remaining base addresses, however, are temporarily holding on registers. Therefore, if process switching occurs, this information must be saved in somewhere corresponding to the process. On the other hand, the *microinterpreter* must have the information required for process management. The process number, the execution priority are examples of it. They are also accessed by the operating system.

Therefore, the information required for process control is divided to two parts, one referred to by both firmware and software, the other accessed only by firmware. The former is called the Process Control Block (PCB), and the later is called the Extended Process Control Block (EPCB).

The built-in predicate "process-control-block(Process,-PCB)" and "set-process-control-block(Process,PCB)" are provided for access to the PCB. However, since EPCB involves significant information for the *microinterpreter* , it is hidden from the software and there is no built-in predicate for accessing the EPCB. When the operating system must access to the EPCB for process creation and process switching, the EPCB is implicitly accessed by the built-in predicates, "initialize-process(Process,StartCode,StackArea)" and "change-process(Process)".

5.3 Interrupt Handling

During KL0 program execution, interrupt signals are checked for at several points in the interpretation process. If one detected, the interrupt interface routine is called. This routine determines the interrupt source, and collects the information related with it. The collected information, such as interrupt code, are saved in the memory by this interface routine. Finally, to activates the corresponding interrupt handler process, process switching is performed automatically at the firmware level.

The built-in predicate, "interrupt-code(Interrupt,Code)", is provided in order to access the saved interrupt information in the activated handler process.

If the entire environment can be saved for restarting the interrupted process, the interruption may be possible at any point. However, it is desirable for fast process switching to minimize the amount of information that must be saved. Therefore, the *microinterpreter* checks for interrupts at the timing after unification and after built-in predicate evaluations. Furthermore, to activate the interrupt handler process as soon as possible, some micro routines which required a long processing time have interrupt check points within them.

5.4 Error Handling

Since there are just two states in general unification rule, "success" and "fail", sometimes illegal conditions are treated as "fail" conditions. Thus, error or exception handling is not sufficiently supported in existing Prolog systems. However, to provide a good programming environment, the system must detect as many illegal conditions as possible. The microprogram implementation can achieve this with little increase in execution time.

If a detected illegal condition is a serious hardware or firmware error, the *microinterpreter* immediately stops execution. If the operating system must deal with it, the *microinterpreter* raises a trap. The trap works in a manner similar to that of interrupts. If an illegal condition is caused by a user program, the *microinterpreter* raises an exception. As described in Section 4.3, 32 system exceptions are defined. Most of them are caused by the illegal input to built-in predicates.

5.5 Protection

Throughout the design of the PSI machine architecture, the protection problem was given low priority. One reason is that PSI is designed for personal use and privacy between users is not considered to be a serious problem. Another reason is that the machine language of PSI is based on logic programming, and, therefore, there is no concept of "address". This means that users cannot access arbitrary memory locations.

However, two protection mechanisms are introduced in order to enhance the reliability of the system. One is the detection of privileged violations and the other is introduction of the new data type having keys.

Although many built-in predicates are provided for users, some must be used very carefully. For example, "set-register" may cause damage to the system. These built-in predicates are considered privileged instructions. The *microinterpreter* restricts their execution within the privileged area, not in the execution mode using in conventional machines. Areas 0 to 3 are defined as privileged areas. When a privileged built-in predicate is called, the *microinterpreter* examines whether it is called from area 0, 1, 2, or 3. If it is not, an exception occurs. By this mechanism, both the *microinterpreter* and the operating system are freed from management of the execution mode, thus simplifying the system.

For privileged data access, KL0 has a special data type, called a "protected type". This type of the data consists of a key and a value. When the protected type appears in unification, it succeeds only if both data are identical; specifically, both arguments pointing physically the same protected type. Therefore, a value of a protected

type cannot be bound during unification. If such binding is necessary, the built-in predicate "set-protected-value(X,Key,Value)" must be used with the correct key. By use of the protected type, the restricted data, such as system directories, can be shared among authorized users who know its key.

6 CONCLUSION

The PSI machine instruction set is designed as high as a logic programming language, Prolog. Its interpreter is implemented completely in microcode. The *microinterpreter* provides high performance and good operating system support by taking advantage of microprogramming techniques and PSI hardware features. It can attain the execution speed of 30K LIPS. A dynamic execution control mechanism and strong error checking are also implemented in microcode.

The kernel of the *microinterpreter* has become larger than we estimated because of powerful KL0 control structures. Its size is about 1.5K steps, half of which are for execution control; the rest are for unification. About 160 built-in predicates are implemented; a third of them support the operating system and low-level hardware manipulations, another third are for data handling. About 10K steps are required for the built-in predicates. Another 1K steps are required for common routines, and for operating system interfaces, such as the interrupt handler.

The basic part of the *microinterpreter* had already been debugged using the microprogram simulator written in Pascal before the PSI hardware was completed. The first version running on PSI was released to the software group on March, 1984.

The microprogrammed implementation enables efficient interpretation of logic programs. However, this approach is not always the best way. A lot depends on the balance between the hardware and software designs. The PSI hardware and the *microinterpreter* are being evaluated in order to clarify the ideal inference machine architecture, and microcode will be refined to show its maximum capabilities based on this evaluations.

ACKNOWLEDGMENTS

The authors wish to express their thanks to all who have contributed to this research, especially to Takashi Chikayama for his advises on the implementation of the KL0 special execution control mechanism, to David H. D. Warren for his valuable suggestions on "argument copying", to Takasumi Ueda for his suggestions on designing the operating system interface. They also would like to express their gratitude to Kazuhiro Fuchi, Director of the ICOT Research Center, and to Toshio Yokoi, Chief of the Third Research Laboratory for their continuous encouragement and support.

REFERENCES

Chikayama, T., Yokota, M., and Hattori, T., *Fifth Generation Kernel Language*, Proc. of Logic Programming Conference '83, Japan, 1983.

Nishikawa, H., Yokota, M., Yamamoto, A., Taki, K., and Uchida, S., *The Personal Sequential Inference Machine (PSI): Its Design Philosophy and Machine Architecture*, Proc. of Logic Programming Workshop '83, Portugal, 1983.

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

Tick, E. and Warren, D.H.D., *Towards a Pipelined Prolog Processor*, Proc. of Int. Symposium on Logic Programming, Atlantic, 1984.

Uchida, S., Yokota, M., Yamamoto, A., Taki, K., and Nishikawa, H., *Outline of the Personal Sequential Inference Machine: PSI*, New Generation Computing, Vol.1, No.1, 1983.

Warren, D.H.D., *Implementing Prolog - Compiling Predicate Logic Program Vol. 1-2*, D.A.I.Research Report, No.39-40, Department of Artificial Intelligence, Univ. of Edinburgh, 1977.

Warren, D.H.D., *An Improved Prolog Implementation which Optimises Tail Recursion*, D.A.I. Research Report, No.156, Department of Artificial Intelligence, Univ. of Edinburgh, 1980.

Yokota, M., Yamamoto, A., Taki, K., Nishikawa, H., and Uchida, S., *The Design and Implementation of a Personal Sequential Inference Machine: PSI*, New Generation Computing Vol.1, No.2, 1983.