# A concurrent multiple-paradigm list processor TAO/ELIS

Ikuo Takeuchi, Hiroshi G. Okuno, Nobuyasu Osato,
Minoru Kamio, Kenichi Yamazaki

Electrical Communications Laboratories
Nippon Telegraph and Telephone Corporation
3-9-11, Midoricho, Musashino, Tokyo, Japan 180

**Abstract**

**TAO is a Lisp dialect with concurrent programming features and other programming paradigms: logic programming and object-oriented programming, implemented on a Lisp machine called ELIS. These paradigms are fused at the core of the evaluation kernel of the interpreter. Hence, the user can mix them in arbitrary granurality without loss of efficency. TAO concurrent programming primitives are so powerful and efficient that several users can enjoy the time-sharing conversation on a simple operating system written in TAO. TAO provides also efficent string manipulation primitives so that Japanese characters and ASCII characters can be mixed in a string. TAO/ELIS is proved to be one of the fastest list processing systems ever made.**

## Introduction

TAO is a Lisp dialect with concurrent programming features and other programming paradigms: logic programming and object-oriented programming, implemented on a Lisp machine called ELIS. This paper describes various topics on the TAO/ELIS system: the overview of TAO/ELIS, the combination of multiple programming paradigms in the language kernel, the concurrent programming, string data types tuned up for Japanese text processing, and the performance evaluation.

## 1. TAO/ELIS Overview

### 1.1 ELIS Lisp machine

The ELIS machine [3] is a 32 bit Lisp dedicated machine. Its CPU is a semi-custom CMOS VLSI which contains about 16K gate logic and 1K bit RAM [10]. ELIS has a 64 bit wide memory bus so that a pair of Lisp pointers, namely a cell, or eight ASCII characters can be read/written in one memory cycle. The ELIS architecture is rather simple compared with those of other Lisp dedicated machines. Table 1 summarizes the specification of the ELIS CPU board. The ELIS CPU board is connected with a front-end processor which interfaces with the secondary storage and terminals. Noticeable features of the ELIS CPU are:

(1) Tagged architecture. Most significant byte of a 32 bit Lisp pointer can be used as a hardware tag.
(2) Fast and large hardware stack, and small register files.
(3) 64 bit Memory General Registers (MGR) which do not only interface between CPU and memory but also serve as general registers. There are a sort of auto-incrementable index registers called SDC which can point to arbitrary byte positions of MGRs. SDC benefits the string manipulation so much.
(4) Large amount of horizontally microprogrammable writable control storage.

**Table 1. ELIS CPU board**

| | |
|---|---|
| size | 29 x 40 cm |
| clock | 16.7 MHz |
| machine cycle | 180 ns |
| internal registers | 32 (32 bits each) |
| MGR | 4 (64 bits each) |
| SDC | 3 (5 or 6 bits each) |
| stack pointers | 3 (16 bits each) |
| hardware stack | 32K (32 bits each) |
| writable control storage | 64K (64 bits each) |
| main memory | up to 128M bytes |
| memory access time | 300 ns |

### 1.2 List processing language TAO

TAO [5, 8, 9] is a Lisp dialect with concurrent programming features and multiple programming paradigms: logic programming and object-oriented programming in addition to the conventional Lisp programming. In short, it incorporates the features of Prolog [11], Smalltalk [2] and the old Flavor [12]. String data types in TAO are tuned up



CH2468-7/87/0000/0167$01.00 © 1987 IEEE

for Japanese text processing. TAO is designed to be an interpreter-centered language to support a fully interactive environment with comfortable high speed. Indeed, the speed of the interpreter gets to be comparable with compilers of other commercial Lisp systems. TAO emulates Common Lisp [7] in the Common Lisp mode.

### 1.3 Programming environment NUE

NUE (New Unified Environment) is to be an integrated programming environment on TAO/ELIS. At present, only a rudimentary and conventional environment has been constructed, which includes an Emacs compatible editor ZEN with Japanese word processing capability, a TCP/IP based network system NAK, a simple window system NOW etc. A truly comfortable and productive programming environment as well as an intelligent office environment is being investigated now.

## 2. Multiple programming paradigms

TAO unifies a Prolog-like logic programming and a Flavor-like object-oriented programming with the conventional Lisp in a slightly extended S-expression syntax. The fundamental mechanisms for these paradigms such as unification, backtracking and message passing are fully microcoded and fused into the Lisp evaluation kernel *eval* smoothly.

### 2.1 Logic programming

The following is an example of the definition of logical append predicate in the logic programming,

```
(assert (lappend (_a . _x) _y (_a . _z))
        (lappend _x _y _z))
(assert (lappend ( ) _y _y))
```

which can be paraphrased to:

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

in the DEC10-Prolog syntax. A symbol prefixed by an underscore denotes a logical variable. It is an independent data type different from normal symbols in order to distinguish logical variables from others in the unification as fast as possible. The *lappend* predicate is treated as a kind of function. The failure of a logical predicate returns *nil* in the Lisp world as its value. Thus, one can use the predicate in the Lisp world as follows:

```
(let (_x y) ... (if (lappend (3 4) _x ,(foo y)) (bar _x) ...))
```

The *if* form tests whether the constant *(3 4)* coincides with the head elements of the evaluated value of the form *(foo y)*, and unifies the logical variable *_x* to its rest elements if successful. If the *lappend* predicate succeeds, that is, if it returns a non-nil value, the instantiated value of *_x* can be used in the Lisp world. If it fails, the value of the *lappend* predicate is *nil* and the value of *_x* is left undefined (uninstantiated). The comma just before a Lisp form (or term in the Prolog terminology) means that the form is to be evaluated before unification. This provides a way to embed the Lisp world in the logic world. In addition, the body of a logical assertion (or sequence of literals in the Prolog terminology) may contain any number of Lisp forms. That is, Lisp forms can be embedded in the logic world. In that case, the value of the Lisp form is interpreted as a failure if it is *nil*, otherwise as a success. Hence, Lisp and logical forms can be mixed in such fine grain. Note that the logical variable *_x* (logical variable with existential quantifier) is declared explicitly in the *let* form here. It is a kind of symmetry between Lisp variables and logical variables.

The internal representation of *lappend* would illustrate more clearly the treatment of logical predicates as functions. The *assert* form above is equivalent to the following form:

```
(define lappend
  (Hclauses
    (&+ ((_a . _x) _y (_a . _z)) (lappend _x _y _z))
    (&+ (( ) _y _y)) ))
```

The function *define* associates a symbol with an anonymous function such as lambda expression. The *&+* expression is a logical counterpart of the *lambda* expression in Lisp, which represents a Horn clause in a "functional" form similar to the lambda expression. The function identifier *Hclauses* denotes another type of function, which represents an ordered set of Horn clauses. The *&+* expression is responsible for the unification and the *Hclauses* expression is responsible for the backtracking. In other words, *&+* corresponds to the logical *and* and *Hclauses* to the logical *or*.

Backtracking in the nested mixture of Lisp and logical forms is somewhat subtle. In TAO, the "scope of backtracking" is limited within the logic world. That is, alternatives in an *Hclauses* are discarded if the logical form which contains the *Hclauses* returns a value to the Lisp world. Side effects caused in Lisp forms which are nested in a logical form are not unwound by the backtracking. However, TAO provides a Lisp function that unwinds the value of logical variables explicitly. In addition, logical variables can be assigned freely in the Lisp world (a mysterious unification from the logic viewpoint!).

It might be confusing that there is no apparent difference between Lisp function calling forms and logical forms. However, it seems natural to see Lisp forms as evaluable predicates or terms from the logic world and to see logical forms from the Lisp world as predicates which may have side effects. Our experience shows that there is little confusion between them. It should be noticed here that one can set the mode whether an "undefined function" causes the error or simply returns *nil* in the logic world, since there are two possibilities: a simple program error and logical failure by the lack of the assertion.

To fuse Lisp and logic worlds, the unification is implemented by the structure copying algorithm. Thus, data structures created by the unification are completely compatible with those created by Lisp. There is no need of data translation between the two worlds.

### 2.2 Object-oriented programming

The object-oriented programming in TAO is, roughly speaking, a successor of the Flavor system in semantics, i.e., with respect to the class hierarchy and method combination. However, the syntax of the message passing is closer to that of Smalltalk. Message passing is represented in a bracketed form (a simple extension of the S-expression):

*[RECEIVER MESSAGE ARGUMENT ...] .*

In the evaluation of the form, *RECEIVER* is evaluated first to determine the receiver of the message. Then, the method corresponding to *MESSAGE* is sought in the class to which the receiver belongs. *MESSAGE* is not evaluated. This form is compatible with the usual infix notation for arithmetic expressions as follows:

*[n + 1]*

where *n* is assumed to be a number. This is the strongest reason why we extend the S-expression syntax to assimilate the Smalltalk syntax. This also illustrates that primitive data types in TAO such as integers and strings are also objects that can receive messages. Thus, one can define a method on the class *integer*. Bracketed lists are of different data type from parenthesized ones. However, only *eval* distinguishes them; most list manipulating functions treat them in the same manner. The execution speed of message passing forms is almost equal to that of the conventional Lisp function calling forms in the current implementation. That is, runtime cost for method search proved not to be a major factor in the (microcoded) message passing mechanism.

Forms for defining a class and its methods are quite similar to those of Flavor and its successors. The class hierarchy allows multiple inheritance. Methods can be combined in various ways as Flavor. However, in TAO, superclass method which may be masked out by the nested inheritance can be invoked explicitly by a super method call as follows:

*(super CLASS.MESSAGE ARGUMENT ...)*

where *CLASS.MESSAGE* is a symbol that concatenates a class name *CLASS* and a message name *MESSAGE* with an intermediate dot. (Note that *super* is a function, not a receiver.) Class variables can be attached to each class. Each class may have class constants, which are a sort of methods that return constant values invariant among all instances of the class. The class constant is introduced because of its fast accessibility and memory economy of instance representation.

The object-oriented programming paradigm is extensively utilized by most system programs, such as the process monitor, terminal system (a sort of termcap), ZEN editor, NAK network system and NOW window system. Especially, Emacs compatible ZEN editor is written fully in the object-oriented style by Yoshiji Amagai et al. It is composed of a number of reusable component classes. Hence, other application programs can use the components of ZEN or ZEN itself as a friendly I/O stream. This facility will enhance the system integrity so much at least with respect to the user I/O interface. The class hierarchy of NAK written by Ken'ichiro Murakami has almost one-to-one correspondence with the TCP/IP layers. It clarifies the program structure of NAK.

However, we feel that the current object-oriented paradigm in TAO is so static that it is often cumbersome to change the characteristics of individual objects dynamically within the framework of the class hierarchy. In a sense, it might be said class-oriented rather than object-oriented. Hence, we are now investigating an extension toward more dynamic object-oriented paradigm where objects can change their characteristics more fluidly.

### 2.3 Object-oriented logic programming

In order to combine the inference power of logic programming and the knowledge modularization capability of object-oriented programming, TAO provides a feature of object-oriented logic programming. This might be useful to construct intelligent knowledge base systems.

A logical assertion can be attached to a class (and its subclasses), which is local to the class(es). For example, one can write

```
(within-class foo
   (assert (&prior _x _y) (left _x _y))
   (assert (&prior red-cube _y)) )
```

which may be read that in the class *foo* something is prior to the other if it is on the left of the other and *red-cube* is prior to anything. To send the message *&prior* to an instance of *foo*, the same form as the usual message passing is used as follows:

```
[x-of-foo &prior _a green-box]
```

where the symbol *green-box* will not be evaluated. The prefix "&" to the print name of a functor serves as an implicit declaration to the compiler that it is a logical message selector in the bracket form. (The interpreter does not care about it, of course.) If the above message passing is done in the logic world, backtracking will take place properly. The method corresponding to a logical assertion is called logical method in contrast with the "procedural" method. Logical methods can be deemed as knowledge local to a class.

Knowledge local or proper to individual instances is represented by a set of unit clauses which contain no logical variables. Each unit clause is called instance fact. Instance facts to a logical method is instance variables to a procedural method. This structural similarity is the key idea of the object-oriented programming in TAO. To assert an instance fact to an object and retrieve the fact, one may write, for example,

```
[x-of-foo &assert like blue-ball]
```

and

```
[x-of-foo & _x blue-ball] or [x-of-foo & like _x]
```

respectively. Note that a desirable subset of instance facts can be retrieved by utilizing the backtracking in the logic world. Of course, the retrieval of instance facts is powered by a sophisticated hashing scheme in the firmware.

The inheritance and method combination in the object-oriented logic programming are parallel to those of the procedural object-oriented programming. Hence, one can exploit the ordering of the inference rules extensively in the class hierarchy.

In addition, TAO provides another kind of combination of unification and message passing. If a user defined object is to be unified with a non-empty list, a message *:unify-next-element* with no argument is automatically sent to the object and the resulted value is again unified with the list. If the object is the value of a logical variable, it is modified to the resulted value of the message passing. The following shows a simple infinite sequence generator, where == is a microcoded unifier defined as: *(assert (== _x _x))*.

```
(defclass generator () (state) () :initable)

(defmethod (generator :unify-next-element) ()
        (cons (inc state) self) )

(assert
 (from _n _seq)
 (== _seq
      ,((make-instance 'generator :state (1- _n))
        :unify-next-element )))

(assert (even-number _x)
      •(from 2 _seq)
       (belong _x _seq)
       (evenp _x) )

(assert (belong _x (_x . _)))
(assert (belong _x (_ . _l)) (belong _x _l))
```

## 3. Concurrent programming

TAO supports a multiple programming and multiple user environment in the framework of the Lisp language. We conceive ELIS to be either a personal AI workstation, time-sharing program developing environment, time-sharing office workstation, or even a back-end Lisp accelerator for commercial machines. We also have a plan to make a parallel computing environment by using multiple ELIS CPU boards. It is crucial for these purposes that TAO could have capabilities of efficient process switching, frequent process interrupts, and flexible process creation and manipulation. In this section, we focus the fundamental Lisp structures of TAO relevant to the concurrent programming.

### 3.1 Scope problem and variable binding

It is a major problem in concurrent programming in the Lisp framework how to control the sharing of the same variable symbol and the sharing of the same variable binding. There may be more than one processes that share the same program body. Some variables are expected to have common values to a set of processes, and others are expected only to have same names. Hence, the scope of variables in TAO is classified in two aspects: locality and sharability, which are not the same as the locality and extent aspects of Common Lisp which does not address the concurrent programming.

Variables are called **local** if they are accessible only within

a lexical context, and called **non-local** if they are declared special and accessible outside the lexical context. In addition, there are variables called **semi-global** which are global within a process but local to the process. That is, the value of a semi-global variable is not visible outside the process. Semi-global variables are a sort of non-local variables, but their bindings are not reset when the process is reset, while the bindings of non-local variables declared as special are of course reset in that case. Thus, semi-global variables can be thought of as user extended instance variables of a process in some sense. (In fact, processes are objects, i.e, instances of the class *process*.) Finally, there are **global** variables which hold values in their value fields. Name conflict of global variables is mainly circumvented by the symbol package facility. (Even if the same user is logged on the same machine twice or more at the same time, the packages corresponding to the login instances differ from each other.)

There is a concrete data type in TAO which represents a sharable variable binding, though it is invisible to the user. The extent of a sharable binding is indefinite. It is, in fact, a specially tagged cons of the form:

*(VALUE . VARIABLE-SYMBOL).*

Non-local variables and semi-global variables use sharable bindings. In addition, a variable (whether local or non-local or semi-global) which is actually closed in a lambda closure are also represented in a sharable binding.

The sharable binding of the non-local variable provides a way to make the variable binding shared among some function call instances in a process. This notion can be easily extended to sharable variables among processes. We use a lambda closure to make variables shared among a set of processes. If a closure is applied in a set of processes, all variables closed in the closure are shared by the processes, since bindings of closed variables are represented in the sharable binding. This is the reason why non-local and semi-global variable can be closed in a lambda closure in TAO, since otherwise variable sharing is necessarily restricted to be local (or lexical) variables and meaningless in many cases. We call such a closure an interprocess closure if it is of the form:

```
(closure '(VAR1 VAR2 ...)
        #'(lambda (fn args) (apply fn args)) )
```

where *VAR1, VAR2, ...* are variables whose bindings are shared. (An interprocess closure can be created by the macro *interprocess-closure*.) If an interprocess closure is set to the instance variable *sys:interprocess-closure* of a process, the process will start by applying the closure to the initial function and initial arguments of the process, instead of simply applying the initial function to the initial arguments. Needless to say, the interprocess closure provides a much finer controllability of variable sharing than simply using global variables for the purpose, since the latter is all or nothing with respect to the sharability range.

The scope of logical variables is extended from that of Prolog's ones, which is limited in a Horn clause where the variables are established. That is, bindings for logical variables may be also represented in the sharable binding. Since the instantiation of a logical variable is a sort of test-and-set operation, logical variables in an interprocess closure can be used for synchronization between processes.

Finally, it is decided in TAO that closed *exit* and *go* are not valid across processes. That is, a closure that contains lexical *exit* or lexical *go* cannot transfer the control back to the original process if the closure is applied in a different process. The reason is that otherwise user's control of processes would be easily lost and there seems to exist no significant consequence of such control transfer that compensates the implementation overheads. However, one can throw to a catch in another process if *process-interrupts* and *throw* are appropriately combined.

### 3.2 Concurrent programming primitives and their usages

TAO provides a variety of concurrent programming primitives without presuming any fixed style of concurrent programming. They seem enough for experimenting the concurrency on a list processing machine. The primitives which are fully microcoded are:

(1) Semaphores with *p-sem* and *v-sem* operations.

(2) Mailboxes with *send-mail* and *receive-mail* operations. The lengths of the mail queue and process queue in a mailbox are, of course, indefinite. There is also a primitive *signal-mail* which actually sends a mail only when at least one process is waiting for mail in the mailbox.

(3) Pipe streams by which a producer process and a consumer process can communicate in a synchronized way. Pipe streams are a kind of bidirectional streams. I/O operations on a pipe stream are the same as those on a normal I/O stream.

(4) Asynchronous process interrupt facility. The function *process-interrupt* sets the status of a process "to-be-interrupted" with given interrupt function and its arguments. The interrupted process will eventually apply the interrupt function to the arguments. The interrupt can be completely transparent to the interrupted process just as the hardware interrupts on the conventional machines.

(5) Two level interrupt inhibition facility. The special form *without-interrupts* executes its body suspending all keyboard interrupts to the process, and *sys:without-interrupts* executes its body inhibiting any process switching unless the process enters waiting status by itself.

Processes, semaphores, mailboxes and pipe streams are all implemented as objects. Processes can be created simply by the function *make-instance*. At the outset, a created process is inert. The function *process-reset* makes the process active, i.e., allocates it in the hardware stack and triggers the function application of the process's initial function to its arguments. If the function application ends, the process becomes inert again. (However, an inert process can be activated temporarily if it is interrupted.) Processes are scheduled by a round-robin discipline with priority queue. Any I/O or event wait can be overlapped with time out wait like *receive-mail-with-timeout*. If mailbox operations and *process-interrupt* are appropriately combined, processes can communicate each other in an asynchronous way. The following is a simple example.

```
(defun process-peep
  (process &optional (fn #'backtrace) args &aux mailbox)
   ; (!x y) is almost equivalent to (setf x y)
   (!mailbox (make-instance 'mailbox :name "peeper"))
   (process-interrupt process
       #'(lambda (m fn args)
                 (send-mail m (apply fn args)))
       (list mailbox fn args)
       t )
   (receive-mail mailbox) )
```

This function asks another process to do something and send back the result. The function application will be done in the environment of the interrupted process. Thus, one can inspect the environment stack of another process without affecting the process by using the defaulted function *backtrace*.

The next example shows the usage of the functions *process-fork*, *wakeup-after* and *process-kill* which are, in fact, all written in TAO itself. The function *idle-process-killer* defined below creates a watching process which watches another process periodically and kills it (and stops itself, of course) if the watched process consumes CPU time below the given threshold in the given interval. Here, *process-fork* simply makes a *process* instance with given initial function and arguments, copies basic environment of the parent process (i.e., dozens of basic non-local variable values) and then resets (i.e., activates) the child process. Note that *idle-process-killer* returns immediately *t*.

```
(defun idle-process-killer
  (process interval threshold
   &optional (fn-before-killed #'(lambda ( ) nil))
             (args ( )) (name "idle-process-killer") )
  (process-fork
   name
   #'(lambda (process interval threshold fn-before-killed
              args &aux cpu-time x )
     (!cpu-time [process :cpu-time])
     (loop (wakeup-after interval)
           (:until (eq [process :whostate] 'stopped))
           (!x [process :cpu-time])
           (:while [[x - cpu-time] >= threshold]
            (process-peep process fn-before-killed args)
            (process-kill process t) )
           (!cpu-time x) ))
   (list process interval threshold fn-before-killed args) )
  t)
```

The last example shows a kind of fork-join which uses a logical variable for synchronization purpose.

```
(defun foo ( )
   (let (_chn)
     (process-fork "client" #'(lambda ( ) (client _chn))
         ( ) :shared-variables '(_chn) )
     (process-fork "server" #'(lambda ( ) (server _chn))
         ( ) :shared-variables '(_chn) )))

(assert (client (_chn1 . _chn2))
        ; i-want is a logical predicate which eventually
        ; instantiates _chn1
        (i-want _chn1)
        (do-something-in-parallel) ; may be Lisp form
        ; wait while _chn2 is not instantiated
        (loop (:while (undefp _chn2) t)
              (process-allow-schedule) )
        (use-the-result _chn2) )

(assert (server (_chn1 . _chn2))
        ; wait until client asks
        (loop (:while (undefp _chn1) t)
              (process-allow-schedule) )
        (welcome _chn1 _chn2) )
        ; eventually instantiates _chn2
```

Note that logical variables _chn1 and _chn2 are not shared by the two processes, and only _chn is shared. Variable _chn need not be declared special in this case since _chn is accessed only within the lexical context. Loops that wait for logical variables being instantiated do not brings in much overheads since process switching in TAO is fast enough as described below.

There can be up to 128 concurrently running (active) processes. Typically, a process switching takes about 40 microseconds. (In worse case, i.e., if switched processes collide in the hardware stack, it may take more than 1 millisecond.) Process switching takes place by the hardware interrupts from the front-end processor, by implicit or explicit process wait and by periodical clock interrupts. Inputs and interrupts from the keyboard are preemptive, so that the user seldom feels delay for his input strokes.

We believe that this is the first successful attempt to provide efficient primitives for practical concurrent programming on Lisp machines. It can be said that the process creation in TAO is much lighter than that of UNIX and that the process switching is much faster than those of other Lisp systems.

## 4. Japanese character string

It is very important for TAO/ELIS to cope with Japanese strings efficiently as well as ASCII strings. However, one Japanese character needs at least 14 bits or 2 bytes (JIS code). There are still hot issues in Japan on how to mix ASCII codes and JIS codes in a single language or in a single string.

TAO solves the problem by regarding a character string as a list of characters rather than as an array of characters, though the characters of a string are packed in an array in the actual implementation. That is, 2 byte JIS codes (with both MSB's set on) and 1 byte (7 bit) ASCII codes are mixed in a string, sacrificing the random accessibility of inner characters. Hence, primitive operations on strings are similar to that of list: *shead* to get a substring at a string head, *stail* to get the tail of a string, *sconc* to concatenate strings, and *snull* to test null string. In addition, one character string is automatically coerced to a character in the native TAO, just as bignum is automatically coerced to fixnum. It is quite contrastive with the conventional array structured string. However, the string representation of TAO seems to give a better way for mixing multi-byte characters in a single string, since it saves memory and it matches the usual left-to-right text processing.

To supplement additional bits to each character in a string, There is a data type called fatstring. It is a cons of two strings: string of additional bits and string of character codes. The representation does not bring in overheads in the normal string manipulation since the additional bits can be removed (or ignored) instantaneously. These extra bits are extensively used in the ZEN editor to support human friendly Kana-Kanji conversion.

## 5. Performance evaluation

We believe that TAO/ELIS is one of the fastest Lisp machines. However, various benchmark tests yield so diverse performance results that we cannot decide exactly how fast TAO/ELIS is. Benchmark tests used in Japanese Lisp community [6] are in favor of TAO/ELIS, while Gabriel's benchmarks [1] are not. However, it is safe to say that the more practical a program is, the faster TAO/ELIS becomes.

For small benchmark tests, TAO/ELIS is as good as other machines, or at least it is not so bad. TAO/ELIS is about two times slower than typical machines for programs that contain many array references, since it does rigorous check for every array access for the present. (Note that the current version of ELIS is not equipped with physical cache memory.) However, the TAO interpreter is faster than compilers on other machines for an over 30K line knowledge base system [4] which runs on various machines. And the TAO compiler is about 5 to 6 times faster than those.

The reasons of this performance figures may be obvious. TAO/ELIS has a fully microcoded interpreter. And more than five hundred system functions are microcoded. Thus, most part of practical programs run actually in microcode.

There may be another aspect of performance evaluation for such machines; overall efficiency of interactive environment. TAO/ELIS may be better in this aspect. In fact, the TAO compiler takes only about 2 minutes to compile about 2000 system functions written in TAO (1.5 mega bytes in source code). The ZEN editor uses a number of microcoded special functions for redisplay, search, and Japanese word dictionary access, thereby several users can enjoy the editor comfortably at the same time. The resulted speed of the TAO interpreter is, of course, one of the most important factors in this aspect. Indeed, we have been using only the interpreter by this spring and seldom felt inconvenience. On the other hand, the speed ratio of the compiler and the interpreter is relatively small; it ranges between 2 and 5. The other merit of the compiler can be found in that the memory consumption by programs is reduced to between 1/3 and 1/5 if compiled.

It should be also emphasized that the compiler is completely compatible with the interpreter, without sacrificing the interpreter interactiveness and the compiler efficiency. The structure of the environment stack is efficient for the compiler and still keeps rigorous information for the interpreter. It enhances the program productivity so much. An example which shows the degree of the compatibility is that even if forms of a function's body are partly compiled and partly uncompiled, the function can work correctly

without any special setting. It was superbly useful in the course of the compiler development. In addition, the backtrace information for the compiled functions gives as fine details as that of the interpreted functions, since all variable names are retained even in the compiled functions. Thus, such queer a utility *disassemble* is scarcely needed.

## Conclusion

TAO/ELIS is unusual in two points. First, it is the first system that combines multiple programming paradigms at the heart of the evaluation kernel as well as concurrent programming facilities. Second, the interpreter achieves high performance comparable with compilers on other Lisp machines. This proves that the interpreter on a dedicated machine can incorporate multiple paradigms and still achieve high performance. It justifies our belief that the interactiveness and runtime efficiency of a system could not be contradicting goals.

We described only part of the TAO/ELIS system in this paper. There are many other unusual features that would comprise a unified environment. At preset, those include:

(1) a powerful assignment mechanism,
(2) computation mechanism based on the address concept like Fortran,
(3) ISAM (Indexed Sequential Access Method),
(4) a binary format of S-expression, called bex, which is ten times faster in read/write operations than normal character files,
(5) the C language which can be incrementally compiled into TAO,
(6) a Japanese Kanji input method compatible with ZEN (or Emacs),

The virtual memory of ELIS is not mentioned in this paper. We are now investigating a new kind of secondary storage usage for truly large applications other than the conventional virtual memory technique. We conceives that TAO is an ever evolving and ever experimental language. It is still changing now as Lao Tse said about 2500 years ago: "The TAO which is called TAO is not the true TAO."

## Acknowledgment

Overall works on TAO/ELIS are not credited only to the titled authors. We thank all persons who contributed to the TAO/ELIS project, especially who contributed to the fundamental of TAO/ELIS: Yasushi Hibino and Kazufumi Watanabe who developed the ELIS machine; Yoshiji Amagai, Tetsuo Ooe and Kazuhiro Chiba who wrote ZEN; Ken'ichiro Murakami who wrote Kermit and NAK; and Takashi Sakai, Shuji Jimbo, Kazunori Yamamori and Hiroshi Gomi who joined us in microcoding. Rikio Onai explored the concurrent programming of TAO extensively. We also thank Prof. Takayasu Ito of Tohoku University for his continual encouragement.

## References

1. Gabriel,R.P. Performance and Evaluation of LISP Systems. MIT Press, Cambridge, MA, 1985.
2. Golberg,A. and Robson,D. Smalltalk-80: The Language and its Implementation. Addison Wesley, Reading, Massachusetts, 1983.
3. Hibino,Y., Watanabe K. and Osato,N. The architecture of Lisp machine ELIS(in Japanese). Report of WGSYM 24, IPSJ, June, 1983.
4. Ogawa,Y., Sugawara,K. and Saito,M. KRINE. Proceedings of the international conference on Fifth Generation Computer Systems (FGCS '84), ICOT,, Tokyo, Japan, October, 1984.
5. Okuno,H.G., Takeuchi,I., Osato,N., Hibino,Y. and Watanabe,K. TAO: A Fast Interpreter-Centered System on the Lisp Mahicne ELIS. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August, 1984, pp. 140-149.
6. Okuno,H.G. The Report of The Third Lisp Contest and The First Prolog Contest. Report of WGSYM 33-4, IPSJ, September, 1985.
7. Steele,G.L. COMMON LISP: The Language. Digital Press, Burlington Massachusetts, 1984.
8. Takeuchi,I., Okuno,H.G. and Osato,N. "TAO - A harmonic mean of Lisp, Prolog and Smalltalk." SIGPLAN Notices 18,7 (July 1983).
9. Takeuchi,I., Okuno,H.G. and Osato,N. "A List Processing Language TAO with Multiple Programming Paradigm." New Generation Computing 4,4(1986).
10. Watanabe,K., Ishikawa,A., Yamada,Y. and Hibino,H. A 32b LISP Processor. Proc. of IEEE International Solid-State Circuits Conference (ISSCC '87), IEEE, New York City, February, 1987.
11. Warren,D.H.D. Implementing Prolog - compiling predicate logic programs. D.A.I. Research Report 39 and 40. University of Edinbourgh, May, 1977.
12. Weinreb,D., Moon,D. and Stallman,R.M. Lisp Machine Manual. LMI, 1983.