Tachyon Common Lisp: An Efficient and Portable Implementation of CLtL2

Atsushi Nagasaka, Yoshihiro Shintani, Tanji Ito
Oki Electric Industry Co., Ltd, Systems Laboratories
11-22, Shibaura 4-chome, Minato-ku, Tokyo 108, Japan

Hiroshi Gomi, Junichi Takahashi
OKI Techno Systems Laboratory
8-10, Uchiyama 3-Chome, Chikusa-ku, Nagoya 464, Japan

Abstract

Tachyon Common Lisp is an efficient and portable implementation of Common Lisp 2nd Edition. The design objective of Tachyon is to apply both advanced optimization technology developed for RISC processors and Lisp optimization techniques. The compiler generates very fast codes comparable to, and sometimes faster than the code generated by UNIX C compiler. Comparing with the most widely used commercial Common Lisp, Tachyon Common Lisp compiled code is 2 times faster and the interpreter is 6 times faster than the Lisp in Gabriel benchmark suit. Tachyon Common Lisp is the fastest among the Lisp systems known to the authors.

1 Introduction

One of the biggest problems of Lisp is its slow execution speed. Current commercial Lisp systems on stock hardwares produce fairly good code using advanced compiler techniques [Brooks 86], [Steenkiste 86]. In the authors' experiences in the development of a Lisp dedicated machine ELIS [Okuno 1984], originally developed by NTT and later by NTT and OKI, which has a microcoded interpreter and a byte-code emulator for compiled code, Lisp dedicated machines now have no advantages, at least in compiled code speed. This is partly due to lack of applicability of this advanced compiler technology and partly to device technology available for special purpose processors.

However, this advanced compiler technology developed for RISC processors has not been fully applied for Lisp systems so far. In Lisp world, the focuses were on Lisp proper optimization techniques such as tail recursion optimization. Lisp has been behind Fortran and C in machine code level optimization, which is required in RISC processors.

The another problem of recent Lisp systems is the slow speed of their interpreters. Since Common Lisp [Steele 84] is a compiler oriented lisp, designers of Common Lisp systems tend to lay emphasis on the speed of compiled code. In addition, since portability or retargetability of a Lisp system is important for implementing a Lisp system on a variety of processor classes, the large portion of the system is coded in a machine independent way, using C and Lisp itself. This results in rather slow interpreters.

One of the strong points of Lisp is its interactive feature and this is one of the sources of the high software productivity of Lisp. The speed of the interpreter is crucial for an interactive environment.

We developed an implementation of the second edition of the Common Lisp language specification [Steele 90] called Tachyon Common Lisp. In addition to the language features defined in CLtL2, Tachyon Common Lisp also provides some extended features which are required in development of application software, including a foreign language interface facility, a graphical user interface such as X-window toolkit and Motif interface, a support of Japanese character set, and a software development environment.

The design objectives of Tachyon Common Lisp are high execution speed of both interpreter and compiled code,

and portability. Our strategy for optimization is to completely apply two kinds of optimization technique; Lisp specific optimization and recent RISC processor optimizations.

In Gabriel benchmarks, Tachyon Common Lisp is 6 times faster in interpreter and 2 times faster in compiler than a widely used commercial Common Lisp. The compiled code is comparable to and for some benchmarks faster than C code.

This paper describes the implementation architecture adopted in Tachyon Common Lisp.

2 Language Kernel

The Language kernel of Tachyon Common Lisp called Plisp is a small lisp which has machine independent specifications to increase the portability of Tachyon Common Lisp. Plisp also supports type specific primitive functions for compiled code entries. We here explain some of the designs adopted in the implementation of Plisp to achieve a very efficient language kernel.

2.1 Implementation of Language kernel

To implement an efficient language kernel, the kernel Plisp was coded in a lisp like macro assembler. Also, the following implementation techniques were used.

Lisp object

A lisp object is represented in 32 bits and consists of pointer part and tag part. The design of the tag scheme in Plisp was developed so as to achieve both fast accesse to lisp objects and a large address space. Tag part is of variable length with 3, 6, or 8 bits including the GC bit. The least significant bit is used as the GC bit. Tag in Tachyon consists of 20 types and a part of them is shown below.

Table 1: Tags in Tachyon Common Lisp (part)

Data Type	Tag assignment
Symbol	01G
Cons	11G
Fixnum	00000G
Simple Vector	00110G
Simple String	00010G
Function	10000G
Single Float	1011110G

By placing the tag part in the lowest position of the lisp object and assigning the tag "000" for fixnum, addition and subtraction of fixnums are executed without extra operations. For cons cell and symbol, tags are assigned so that lisp objects point to their object bodies. This assignment made it possible to access cons cells and symbols without tag masking operations.

Type specific data structure

Vectors and arrays in Common Lisp have general powerful features such as fill pointers and adjustable arrays. These features make compiler and interpreter slow. Tachyon Common Lisp provides simple vectors and simple arrays, separately implemented from general vectors and arrays.

Register convention and argument passing

In RISC philosophy, it is important to decrease the number of memory accesses, provided that a large register set is given.

7 of 32 general registers are used as global registers. These variables are permanently placed on the registers and never saved onto stack, which reduces stack access operations. Tachyon Common Lisp uses 8 registers for parameter passing in function calling. In the case of more than 8 arguments, the first 8 arguments are placed in registers and and the rest of the arguments are placed on the stack.

Table 2: Register Usage in Tachyon/i860

Register	Usage
r0	All Zero
r1	Return Address
r2	System stack pointer
r3	Lisp frame pointer
r4	Lisp stack pointer
r5	Environment frame pointer
r6	System mode
r7	Global pointer
r8	NIL
r14	Address of &rest variables
r15	No of arguments/return values
r16 - r23	Arguments/Return values
rest	Work registers

Multiple values are also returned in registers in the same way to decrease memory accesses.

Stack overflow check

Using the page-wise memory protection facility of UNIX System V R4, stack overflow checks in stack accesses can be removed, which significantly improves the speed of stack operations. The memory page being placed above the stack boundary is memory protected. Memory accesses exceeding the boundary cause a memory protection fault. Each stack access does not require a stack overflow check.

Global variable table

In i860 [Intel 89] and also many other RISC CPUs, loading a datum into a register requires 2 instructions as shown below.

```
orh     address@ha,     r0,     r31
ld.l    address@l(r31),         r31
```

For reducing the overhead in variable references, global variables defined in Common Lisp and variables that are frequently accessed by the interpreter are located together on a global data table. Using a register which points to the top of the table, these variables are accessed in one instruction.

```
ld.l    offset(r7),             r31
```

2.2 Type specific functions

One of the features of Common Lisp is its type generic operations represented by sequence functions. To reduce the overhead of run-time type checking, Plisp provides type specific functions and functions with fixed number arguments especially for compiled code.

When type and number of arguments is known to the compiler at compilation time, the compiler produces code that calls these type specific Plisp functions.

2.3 Lisp style macro assembler LT

To implement a fast language kernel and to improve the portability of Plisp, Plisp was coded in a lisp like assembler LT (short for Lisp Translator) which was newly developed by the authors. Unlike ordinary macro assemblers, an LT program looks like a Lisp program. LT keeps both the complete access to hardware capability and the high readability of Lisp. Only 2 % of Tachyon Common Lisp was coded in C and the rest was coded in LT and Common Lisp in a metacircular way.

The features of LT are given below.

- LT can access all hardware capabilities.
  Users can assign variables to registers so that register saving and restoring overheads are minimized. Inside Plisp, virtually no register saving or restoring in function call is needed.
- Syntax of LT is close to that of Common Lisp.
  LT has only 20 original constructs and can be easily extended using Common Lisp macro facility. (LT itself was developed in Common Lisp) Using this extendibility, LT allows programs to be portable and highly readable.
- LT supports delayed branch
  Combining a delay-expression of LT and macro facility, branches can be used without detailed knowledge of the target processor. An example of delayed branch is given below.

```
(defmacro goto (entry)
 '(delay (br ,entry)))

(setq r4 1)                 br   label
(goto label)                addu 1, r0, r4
```

An example of LT coding is given below.

```
(defmacro WHILE (c i &rest body &aux label)
     (setq label (create-temp-label))
     '(block nil
          ,label
          (if (not ,c) (return))
          ,@(cons i body)
          (goto ,label)))
(defmacro CDR (x) '(word ,x -2))
(defmacro CONSP (x &optional (tmp x))
     '(progn (setq ,tmp (and ,x #b110))
             (= ,tmp #b110)))

;; Function length returns the length
;; of a linear list.
(defun length (list &aux tmp count)
     (setq count 0)
     (WHILE (CONSP list tmp)
            (setq count (+ count 1))
            (setq list (CDR list)))
     ....
     )
```

2.4 Memory management

The ratio of time consumed by garbage collection in program execution is large and suspending program execution caused by GC is not desirable for applications. Tachyon Common Lisp employed Ephemeral GC as its memory management scheme.

Heap memory space consists of a list of memory regions, which are allocated as needed. A Lisp object is first allocated in the youngest region. As the existing time period of an object gets longer, the object is moved to the older regions by GC. The memory manager usually executes copying GC for the youngest regions and executes compactifying GC for older regions in the direction of the oldest.

A feature of the GC of Tachyon is that the number of regions for the copying GC is dynamically varied depending on the amount of garbage collected in the last GC or on some memory usage estimate. This improves the efficiency of GC.

3 Intermediate code: Lcode

3.1 Register machine

Lcode, the intermediate code for Tachyon Common Lisp compiler, is based on the register machine model be-

cause Tachyon Common Lisp is targeted on computers with a RISC processor. Lcode is designed so that Tachyon Common Lisp runs efficiently on RISC processors with a large set of registers(usually 32 registers). As shown, the model is simple and generally reflects the common features of RISC processors, which contributes to its high portability and efficiency.

Stack

Lcode has a virtual stack for placing function frames and data on. The actual implementation of Tachyon Common Lisp on i860 has two stacks, Lisp stack and system stack. However, system stack does not appear in Lcode.

Registers

Lcode assumes that a processor has a set of general purpose registers, and the uses of some of the registers are predefined as shown below.

- Arguments
- Return values
- Number of return values
- NIL
- Stack pointer
- Function body

3.2 Lcode primitives

Lcode consists of about 40 primitives. Primitives are classified in the following categories;

(1)	Function interface	(6)	Stack operation
(2)	Argument check	(7)	Branches
(3)	Multiple value	(8)	Non-local exit
(4)	Function creation	(9)	Error handling
(5)	Data transfer	(10)	Vector operations

Each primitives has an attribute code in its argument. An attribute code provides information obtained in the program analysis phase of the compiler, such as type of arguments, recursion instruction, current stack depth. This information is used in the code generation phase and helps the code generator to produce efficient codes.

4 Optimizing Compiler

The optimization strategy in Tachyon Common Lisp is to exhaustively apply the following two kind of optimization techniques, (1) Lisp proper optimization techniques such as tail recursion optimization and program unfolding and (2) recent optimization techniques developed for the RISC processor such as instruction scheduling and register allocation using the large register set.

4.1 The structure of Compiler

The compiler consists of the following phases.

1. Preprocessing phase
   - Macro expansion and inline expansion of flet labels and so on.
2. Syntax analysis and Variable analysis
   - Syntax analysis
   - Type inference for variables
   - Constant folding etc
3. Translation to upper level intermediate language
   - Translates functions into more primitive functions: ex. map functions
   - Removal of closures and cell consuming functions
4. Translation into Lcode
   - Translation into Lcode
   - Global/local optimization
5. Translation into Assembly code
   - Inline substitution
   - Local optimization
6. Translation into machine code
7. Loading onto memory

4.2 Optimization techniques

In addition to the following basic optimization techniques, Tachyon applies the more advanced optimizing methods described below.

Lcode	Machine code
(1) Removal of closures	(1) Remove Non-reachable code
(2) Removal of cell consuming function	(2) Remove push/pop pair
(3) Remove double jump	(3) Remove double jump
(4) Remove push/pop pair	(4) Direct call
(5) Constant folding	(5) Branch optimization
(6) Branch optimization	(6) Instruction schedule
(7) Register allocation	
(8) Tail recursion optimization	

Code generation through Data type inference

The absence of type declaration for variables is the cause of inefficiency of Lisp, compared with other languages. Data type inference has been studied and applied for code optimization [Shivers 91]. Tachyon Common Lisp also applies the type inference techniques to improve compiled code speed. The Compiler has a database on the language specifications of Common Lisp. The database contains the information on data type of arguments and return values for each Common Lisp function and each Tachyon internal function including Plisp. If the data type of a variable is determined, the compiler generates type specific operations for the variable. In addition, since Plisp provides type specific functions and functions with a fixed number of arguments, the

overhead in the run-time libraries is minimized.

[Steele 78] showed that with knowledge on the language constructs, a compiler can produce better code. The compiler also knows the specifications on control for primitive functions (and special forms and macros). Using this knowledge, the compiler analyzes programs to produce more efficient code.

- Type specific functions
  If the data type of arguments is determined, the compiler produces a code which calls the type specific functions.

  Example 1:

```
(delete item list)    ;; item, list are lists
      ⇓
(delete-list item list)
```

  Example 2:

```
(setq x (length y))   ;; Function length returns
    ....              ;; a fixnum.
(1+ x)                ;; The compiler generates
                      ;; fixnum addition.
```

  The compiler generates a fixnum addition followed by `intovr` instruction. If the fixnum addition causes an overflow, `intovr` detects the overflow and the result is converted into a bignum in the overflow handler.
- Type of Arguments
  If data type of arguments is determined, type checking code can be removed and the compiler produces code which call a function with no type checking.
- Multiple values
  If the number of return values of a function is known, processing of multiple values will be simplified. The next is the case of (multiple-value-setq (x y z) (floor a b)) with 2 values.

  Example 3:

```
(floor a b)                          (floor a b)
if number of values > 0              x = value1
x = value1 or nil            ⇒       y = value2
if number of values > 1              z = nil
y = value2 or nil
if number of values > 2
Z = values3 or nil
```

Direct Function Call

In the direct function call [Zorn 90], a caller function directly passes the control to the entry point of a callee function to reduce the overheads in regular function calls through function symbols. The problem in direct function call is the redefinition of callee functions. Tachyon handles this problem by maintaining the information on the calling functions.

Specialization

The optimization, based on the knowledge of function specifications and about specialization by function combination, is applied in the compiler. As an example, for the inline substitution of a test-function in a conditional branch, if the return value of the function is not required, then the compiler produces a code to branch without creating the return value.

Inline substitution of functions

Lisp primitive functions are inline substituted to reduce the function calling overheads. User defined leaf functions are also inline expanded according to the optimization declaration.

Instruction Scheduling

Instruction scheduling for delayed slot and register assignment techniques for a large set of registers developed for RISC processors are used. Intel i860 has the three types of delayed slots; delayed jump, delayed branch after operation (a delayed slot for a conditional branch immediately after an operation) and delayed load. The compiler schedules instruction streams to fill these delayed slots.

5 Portability

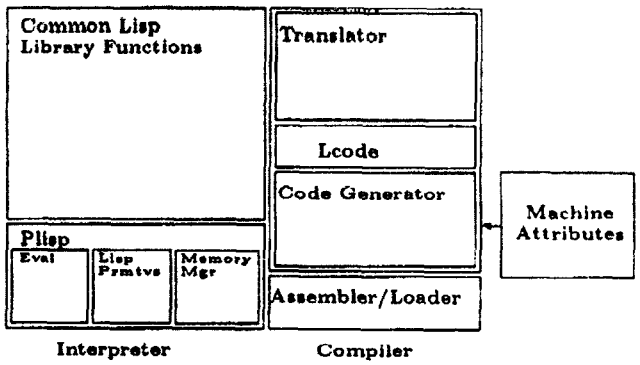
Efficiency and portability are conflicting targets. To achieve the portability, Tachyon separated the machine dependent part from the machine independent part. Figure 1 shows the structure of Tachyon Common Lisp. Only the macro definitions for low level primitives in Plisp and Machine attributes file are machine dependent.

For portability, Tachyon adopts the following design schemes.

- Clear separation of machine dependent and independent part.
- Implementation of Plisp in a Lisp like macro assembler LT. Using macro facility of LT, porting Plisp onto a different processor can be easily done by rewriting machine dependent macros.
- Machine independent intermediate code Lcode and Machine attribute file
  Lcode is machine independent intermediate code and the code generator produces machine code by referring the information described in the Machine attribute file.

Tachyon Common Lisp was first implemented on OKIstation7300 and later ported onto SparcStaion2. The porting of the interpreter and the compiler required about 2 man-months and 2 man-weeks respectively. The

Figure 1: Structure of Tachyon Common Lisp



porting of the interpreter included redesign of internal structure of Plisp for improving the portability and reimplementation of LT. This shows that Tachyon Common Lisp certainly has high portability.

6 Performance

Table 3 compares Tachyon Common Lisp on OKIstation7300 against Lucid Common Lisp (Version 3.0) on SparcStation 2 in interpreter for Gabriel benchmark. Tachyon Common Lisp on i860 is 6.02 times faster than Lucid Common Lisp after normalizing hardware performance with SPECint. The superiority of Tachyon Common Lisp interpreter to Lucid Common Lisp mainly comes from the efficient implementation of Plisp.

Table 3: Gabriel Benchmarks/Interpreter (sec)

Benchmark	Tachyon CL/i860	Lucid CL
Tak	1.589	9.950
Stak	3.031	19.210
Ctak	2.317	12.250
Takl	12.671	110.160
Takr	1.918	11.440
Boyer	26.600	135.820
Browse	39.910	189.750
Destructive	7.350	46.090
Traverse-init	53.700	750.790
Traverse-run	267.400	2077.630
Derivative	4.090	22.130
Dderivative	4.940	19.870
Div2-iter	5.480	38.900
Div2-rec	4.520	21.740
FFT	2.170	10.390
Puzzle	40.340	231.630
Triangle	480.370	3486.160

Table 4 shows the comparison of Tachyon Common Lisp, Lucid Common Lisp, and CMU Common Lisp (Version 15a) [MacLachlan 91],[Fahlman 91] in the compiled code for the same benchmark. In the table, Tachyon/i860 is the version for OKIstation7300 and Tachyon/Sparc is the version for SparcStation2. Lucid-Development represents Lucid development mode compiler and Lucid-Production represents Lucid production mode compiler. "ratio" means the relative speed against Lucid production mode compiler. Tachyon/i860, Lucid Common Lisp and CMU Common Lisp all runs on the same SparStation 2.

Each programs was compiled with the same optimization option; speed=3, safety=0, space=0, and compilation-speed=0 except for Lucid development mode compiler(compilation-speed=2). The ephemeral GC is disabled in Lucid Common Lisp. The Sparc version of Tachyon is not fully optimized yet and does not use the register window architecture of Sparc. The table shows that Tachyon Common Lisp is 2.01 times faster than Lucid production mode compiler and 3.53 times faster than Lucid development mode compiler. Both are geometric means over the benchmarks.

Figure 2 is the graph of Table 3. The two graphs of Tachyon/i860 and Tachyon/Sparc are of the similar figure. Speed improvement of Tachyon/Sparc over Tachyon/i860 in Takr is due to the size of cache memories.

Table 5 is shows the effects of each optimization techniques. Optimization techniques are divided into the three categories; Inline substitutions, Direct function calls, and Instruction scheduling including peephole optimizations. The effect of type inference is not given here because its code is spread out over the compiler and it is difficult to measure its effect. In the table, "All" means that all the optimizations are applied. "None" means that none of the above three is applied, and "All $\ominus$ Direct call" means that direct function call is not applied.

The optimization techniques are listed in order of effectiveness; Instruction scheduling and peephole optimization (30%), Inline substitution (20%) and Direct function call (14%). It should be noticed that these values are geometric mean over the benchmark suit and that each optimization technique is not independent from the others. For example, direct function call doubles the speed of Boyer but does not effect to Tak, though both are function call intensive programs, because tail recursion optimization gives the same effect.

Even without applying the three optimization techniques, the speed of Tachyon compiler is still comparable to Lucid production mode compiler. It is concluded that type inference and other optimization techniques including the implementation of Plisp greately contribute to the efficiency of Tachyon Common Lisp.

Table 4: Common Lisp Performance Comparison

Benchmark	Tachyon/i860		Tachyon/Sparc		Lucid-Development		Lucid-Production		CMU Common Lisp	
	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio
Tak	0.051	0.90	0.041	1.12	0.100	0.46	0.049	1.00	0.120	0.38
Stak	0.243	3.47	0.287	2.94	1.020	0.83	0.855	1.00	0.300	2.81
Ctak	0.170	1.63	0.189	1.49	0.420	0.67	0.292	1.00	0.449	0.63
Takl	0.115	1.78	0.122	1.68	0.640	0.32	0.200	1.00	0.346	0.59
Takr	0.115	0.45	0.045	1.44	0.100	0.65	0.069	1.00	0.232	0.28
Boyer	0.770	2.23	0.660	2.61	2.950	0.58	2.331	1.00	2.250	0.76
Browse	0.460	5.17	0.480	4.98	4.190	0.57	3.101	1.00	7.790	0.31
Destructive	0.198	1.40	0.160	1.74	0.480	0.58	0.389	1.00	0.465	0.60
Traverse-init	1.281	1.12	1.387	1.03	5.430	0.26	1.425	1.00	2.420	0.59
Traverse-run	3.178	1.15	3.207	1.24	20.740	0.19	4.045	1.00	4.970	0.80
Derivative	0.280	1.64	0.270	1.70	0.740	0.62	0.565	1.00	0.450	1.02
Dderivative	0.390	1.51	0.320	1.84	0.860	0.69	0.633	1.00	0.680	0.87
Div2-iter	0.170	1.41	0.170	1.41	0.320	0.75	0.264	1.00	0.260	0.92
Div2-rec	0.150	8.80	0.160	8.25	1.170	1.13	1.114	1.00	0.380	3.47
FFT	0.109	2.66	0.159	2.09	0.490	0.59	0.397	1.00	0.640	0.45
Puzzle	1.430	2.61	2.301	2.54	4.590	0.81	4.066	1.00	10.900	0.34
Triangle	16.554	1.78	14.398	2.05	42.000	0.70	32.220	1.00	106.490	0.28
Geometric mean		1.98		2.01		0.57		1.00		0.66

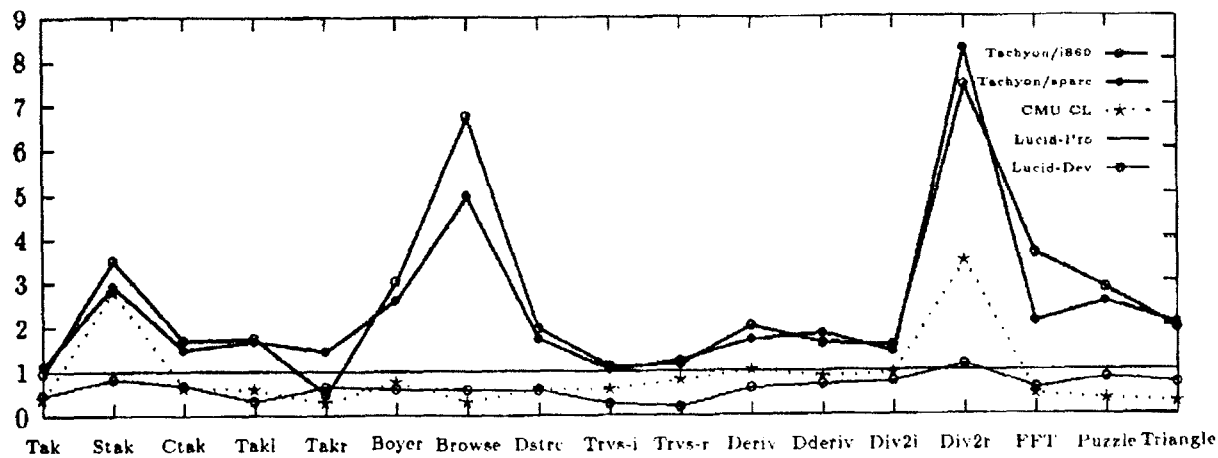


Figure 2: Common Lisp Performance Comparison

Table 5: Effect of Optimization Techniques

Benchmark	All		All ⊖ Direct call		All ⊖ Inline		All ⊖ Schedule		None	
	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio	time(sec)	ratio
Tak	0.051	1.00	0.051	1.00	0.700	0.73	0.090	0.57	0.109	0.47
Stak	0.243	1.00	0.243	1.00	0.288	0.84	0.261	0.93	0.306	0.79
Ctak	0.170	1.00	0.170	1.00	0.188	0.92	0.213	0.81	0.232	0.74
Takl	0.115	1.00	0.196	0.59	0.115	1.00	0.443	0.26	0.524	0.22
Takr	0.145	1.00	0.254	0.57	0.152	0.95	0.216	0.67	0.338	0.43
Boyer	0.770	1.00	1.610	0.48	1.070	0.72	1.150	0.67	2.480	0.31
Browse	0.460	1.00	0.470	0.98	1.220	0.38	0.610	0.75	1.590	0.29
Destructive	0.198	1.00	0.198	1.00	0.269	0.74	0.265	0.75	0.363	0.55
Traverse-init	1.281	1.00	1.424	0.90	1.741	0.74	1.705	0.75	2.342	0.55
Traverse-run	3.178	1.00	3.477	1.00	3.498	0.99	5.235	0.66	5.246	0.66
Derivative	0.280	1.00	0.410	0.68	0.360	0.78	0.360	0.78	0.580	0.48
Dderivative	0.390	1.00	0.490	0.80	0.480	0.81	0.420	0.93	0.760	0.57
Div2-iter	0.170	1.00	0.170	1.00	0.180	0.94	0.210	0.81	0.240	0.71
Div2-rec	0.150	1.00	0.150	1.00	0.190	0.79	0.210	0.71	0.230	0.65
FFT	0.109	1.00	0.114	0.96	0.120	0.91	0.132	0.83	0.149	0.73
Puzzle	1.430	1.00	1.478	0.97	1.683	0.85	1.884	0.76	2.187	0.65
Triangle	16.554	1.00	16.598	1.00	21.682	0.76	25.138	0.66	29.924	0.55
Geometric Mean		1.00		0.86		0.80		0.70		0.52

For performance comparison of Tachyon Common Lisp and C, some benchmark programs in Lisp and C were run on OKIstation7300. Lisp programs were converted into the equivalent C programs and C programs were rewritten in Common Lis. Type declarations are given for Lisp programs.

Table 6: Comparison Tachyon and C

Benchmark	Tak	Takr	Sieve
Tachyon/i860	0.031	0.083	0.018
Tachyon/Sparc	0.024	0.033	0.017
PCC/i860	0.047	0.105	0.010
SUN C	0.078	0.084	0.022
SUN C with - O	0.026	0.041	0.007

Table 6 shows the result of the comparison. In comparison with C, Tachyon is faster than C code in some benchmarks such as Tak. As shown in Sieve, array references and loop constructs of Lisp are slower than C, but they can be improved for typical simple arrays.

7 Concluding Remarks

An efficient and portable Common Lisp based on the CLtL2 language specification was developed. Comparing with other Common Lisp systems, Tachyon Common Lisp is 2 or more times faster and its code is comparable to and sometimes faster than C code. Tachyon Common Lisp is the fastest among the Lisp systems known to the authors. In the current version of Tachyon Common Lisp, machine code level optimization and some of the planed optimization techniques have not been finished and there is a room to improve the performance.

Tachyon Common Lisp shows that there are many opportunities to improve the performance of Lisp systems by applying the advanced compiler techniques developed for RISC processors and Lisp specific optimizations such as type inference.

References

[Brooks 82] Rodney A. Brooks, Richard P. Gabriel, and Guy L. Steele Jr. An Optimizing Compiler for Lexically Scoped LISP, *Proceedings of the 1982 ACM Conference of LISP and Functional Programming*, pp.261-275, 1982

[Brooks 86] Rodney A. Brooks, D.B.Posner, J.L.McDonald, and J.L. White. Design of An Optimizing, Dynamically Retargetable Compiler for Common Lisp, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp.67-85, 1986

[Fahlman 91] Scott E. Fahlman and David B. McDonald. Design Considerations for CMU Common Lisp, *Topics in Advanced Language Implementation*, pp.137-156, MIT Press, 1991

[Gabriel 85] Richard P. Gabriel. "Peformance and Evaluation of Lisp Systems." MIT Press, 1985

[Intel 89] Intel Corporation. *i860 64-bit Microprocessor Programmer's Reference Manual*, Intel Corporation, 1989

[MacLachlan 91] Robert A. MacLachlan. CMU Common Lisp User's Manual, CMU-CS-91-108, 1991

[Okuno 1984] Hiroshi G. Okuno, Ikuo Takeuchi, Nobuyasu Osato, Yasushi Hibino, and Kazufumi Watanabe. TAO: A Fast Interpreter-Centered Lisp System on Lisp Machine ELIS, *Conference Record of the 1984 Symposium on Lisp and Functional Programming*, pp. 140-149, August 1984, ACM

[Steele 78] Guy L. Steele Jr. "RABBIT: a Compiler for Scheme." MIT AI-TR No. 474, 1978

[Steele 84] Guy L. Steele Jr. Common Lisp: the Language, Digital Press, 1984

[Steele 90] Guy L. Steele Jr. Common Lisp the Language, 2nd Edition, Digital Press, 1990

[Steenkiste 86] P. Steenkiste and J. Hennessy. LISP on a reduced-instruction-set-processor. *Proceedings of the 1986 Conference on Lisp and Functional Programming*, pages 192-201, August 1986, ACM

[Steenkiste 91] P. A. Steenkiste. The Implementation of Tags and Run-Time Type Checking. *Topics in Advanced Language Implementation*, pp. 3-24, MIT Press, 1991

[Shivers 91] Olin Shivers. Data-Flow Analysis and Type Recovery in Scheme, *Topics in Advanced Language Implementation*, pp.47-87, MIT Press, 1991

[Takeuchi 86] I. Takeuchi, H. Okuno and N. Ohsato. A List Processing Language TAO with Multiple Programming Paradigms, *New Generation Computing*, Vol. 4, No. 4, pp.401-444, 1986

[Zorn 90] B. Zorn and P. Hilfinger. Direct Function Calls in Lisp, *LISP AND SYMBOLIC COMPUTATION: An International Journal*, 3, pp.13-20, 1990