Regular Paper

A Concurrent Real-time Garbage Collector

Ikuo Takeuchi,[†1] Yoshiji Amagai,[†2] Masaharu Yoshida[†3]
and Kenich Yamazaki[†4]

In this presentation, we report on the microprogramming implementation and its evaluation of a concurrent real-time garbage collector (GC) for the rea-time symbolic processing system TAO/SILENT. Our real-time GC is implemented as a set of GC processes which run concurrently with other Lisp processes. It is based upon a sort of well-known incremental update algorithm, and it brings in only a little overhead to Lisp primitives. We embedded a scheduling mechanism that is specific to GC process scheduling, since it is indispensable to devise a good cooperation scheme between the operating system kernel and concurrent GC. We also developed a good deal of tiny techniques to make the GC processes as swift as possible. As a result, our concurrent GC achieves a very small response delay for external events, that is, when an interrupt event takes place, the corresponding urgent Lisp process can wake up in less than 131 microseconds for the worst case where more than ten thousand memory-consuming Lisp processes are concurrently running, and surely in less than 50 microseconds if those Lisp processes are made with real-time consciousness. These figures are more than a hundred times as good as most of those appeared in the literature. Our GC can be said "transparent" in the sense that real-time processes would not be aware of the GC whether it runs or not, at least with respect to response delay.

1. Introduction

The necessity of real-time garbage collector, or simply real-time GC[☆], that is bearable to hard real-time applications has long been advocated in literature. A lot of real-time GC techniques have been developed for a variety of contexts and problem domains [1)~16)]. However, to the authors' knowledge, a few of them seem truly applicable to hard real-time applications that need microsecond order response time. Hard real-time systems that involve symbolic processing have been classified as a big challenge [17)].

Hard real-time symbolic processing has to assure that no given deadline is ever violated, where "deadline violation" may include too early fulfilling before deadlines; nothing special is added in the context of symbolic processing. Note that, in an application, there may coexist real-time processes that should meet deadlines strictly, and non real-time, or at least, soft real-time processes whose failure in meeting deadlines is not fatal. We use the word *real-time process* for those which have severe mission for real-time response.

In the literature of Lisp languages, the GC is often called an obstacle for Lisp to get being widely used in real-time systems. The pause time caused by the GC is only the first half of the response delay, and the computation performed by a real-time process before it actually responds to an event should be taken into account for the second half of the response delay. We will call the first half a wake-up delay (of the real-time process) in the paper. It is still a big problem how to accommodate in real-time systems sophisticated symbolic processing which takes unpredictable time[☆☆]. This problem is, however, related to a sort of pragmatics, outside the scope of a real-time symbolic processing system design. But it is, at least, important to guarantee the performance figures of symbolic processing primitives, with which users can deliberately design their real-time application profile. That is, if a real-time GC imposes much overheads on these primitives, a real-time process suffers from the poor performance for its actual computation even if it is waken up very quickly by an urgent event.

A real-time symbolic processing system should have the following four features:

- powerful multiprogramming capability,

†1 The University of Electro-Communications
†2 NTT
†3 NTT-IT
†4 NTT DoCoMo

☆ Hereafter we occasionally use the word GC for both garbage collection and garbage collector.

☆☆ This is the reason why we do not dare to call our system hard real-time.

- interrupt handling facilities that can handle external events and internal clocks etc.,
- no long inseparable sections inside the system, and
- real-time garbage collector.

There may be long inseparable sections in some sophisticated symbolic processing primitives. For example, automatic rehashing of a hash table equipped in Common Lisp may be a long inseparable section; that is, in the course of rehashing, no process would be able to access to the hash table if it is implemented without being aware of real-time processing. Such inseparable sections should be broken down into as fine fractions as possible, allowing other processes quite often interleave with them without destroying the system integrity. We will call such breaking down "grinding up an inseparable section." For the worst case, at least, use of lengthy inseparable sections should be restricted by the runtime system parameters for serious real-time applications.

The GC also may contain long inseparable sections inside. The time needed for those is called pause time in the literature, because no proper symbolic processing can be done for its duration. In a real-time GC, at least, on a single processor system, these inseparable sections should be ground up into finer program segments, thereby they can be interleaved quite often with other processes, called mutators, that consume and modify data objects. In other words, the GC should be incremental because it is impossible to finish all the collection at once. In addition, the GC should have a reasonable speed of collecting unused (unreferenced) data objects so that it can always supply enough fresh memory to mutators.

In this paper, we describe the GC implemented by microprogram on the real-time symbolic processing system TAO/SILENT developed by us. As we explain later, this GC is in a sense almost *transparent* to real-time processes.

2. TAO/SILENT

Before describing our GC, we introduce the TAO/SILENT system very briefly because some of its features are closely related to the GC.

SILENT whose CPU LSI was fabricated in 1993 is a dedicated symbolic processing machine, or it may be simply called a Lisp machine, which is a direct successor of the ELIS Lisp machine that was also developed by us in mid 80's [18]. TAO [19] is a symbolic processing language based on Lisp with multiple programming paradigms such as object-orientation and logic programming. TAO is also a successor of the multiple paradigm language TAO [20] on the ELIS machine. It inherits the name, but it is entirely re-designed to be more elegant and efficient with respect to real-time programming. But it is enough in this paper to consider TAO as a Lisp dialect equipped with a rich variety of data types and multiprogramming capability.

SILENT is a micro-programmable 40 bit word machine; 8 bits for tag, and 32 bits for pointer or immediate data. The MSB (most significant bit) of the car's 8 bit tag is used for the GC mark bit. SILENT has 80 bit wide bus to read/write a cons cell at once.

SILENT is equipped with a 256 K word hardware stack memory. For each 128 word block, there is a 2 bit dirty flag, each bit of which is set to 1 when something is written in a word within the block. We will call the 128 word block a *D-block*.

The SILENT machine cycle is 30 nanoseconds; that is, the system clock is 33 MHz, which is considerably slow compared with the current microprocessors on the market. For the sake of real-time processing performance, it has no virtual memory mechanism; that is, SILENT is a real memory machine.

The operating system kernel, or micro-kernel, is written fully in microcode. It can control up to 64 K concurrent processes. The shortest lifetime of a process, that is, the minimum time needed to create a process, give it a null task to run, and leave it to stop, is 17 μsec. The shortest possible process switching time is 4 μsec [21].

Processes can have a priority from 0 to 63 and a protection level from 0 to 3, where bigger number corresponds to higher priority and protection level, respectively. The micro-kernel makes use of a preemptive priority-based scheduling with round-robin CPU rotation.

One of the important jobs of the micro-kernel is to manage the efficient usage of the hardware stack; that is, it has to swap in or out a process's stack between the hardware stack memory and the main memory when processes' stacks are going to collide.

For the stack management, four contiguous D-blocks are grouped as a stack block. In a stack block, only one process's stack can exist. Thus, roughly speaking, if the sum of all

active processes' stack sizes is reasonably less than 256 K words, all of them can coexist on the hardware stack memory and enjoy fastest process switching.

There are, however, a few long inseparable sections in the micro-kernel, mainly for swapping stacks. Long ones take about 70 μsec at maximum. It is important to note here that we assume that real-time process does not use much stack, and thereby a real-time process itself will not load the micro-kernel down so much.

3. Basic GC Strategy

The basic strategy of our GC is characterized by the following four features:

- incremental update with write-barriers,
- mark-sweep,
- no compaction, and
- a set of very concurrent GC processes.

The details of our incremental update write-barrier algorithm will be described in Section 4. Roughly speaking, write-barriers check every pointer modification in order to protect newly written reference from being out of the GC's sight, when it is to be written in a data object that the GC has already traversed in a mark phase. Write-barriers notify the GC of the overwriting reference that would otherwise be collected as garbage.

As was depicted in 22), incremental update write-barrier algorithm is better in various aspects than snapshot-at-the-beginning write-barrier algorithm [14]. For example, it is less conservative, i.e., more garbages are likely to be collected, and thereby the cost of traversing readily to-be-garbage objects is reduced. Write-barriers are also better than read-barriers, since modifying data objects is less often than dereferencing pointers in most application programs.

In the context of an incremental update GC, copying method, an alternative to mark-sweep method, is more often adopted in the present real-time GCs. There may be pros and cons to both methods in various conditions, but mark-sweep is obviously more appropriate for real memory machines like SILENT.

Compaction is desirable if memory fragmentation is a serious concern. However, we did not incorporate it in the current implementation because it would deteriorate real-time performance of our GC and we estimated that the fragmentation problem is not so serious in our real memory environment.

A remarkable feature of our GC is that it consists of eight processes that run concurrently with mutators. We have two marking processes *main* and *post* which run complementarily in a mark phase, and six mutually independent sweeping processes each of which corresponds to a data type category such as cell (two field data unit), vector (data unit of arbitrarily variable size) and buddy (system data unit of 2's power size) ☆.

The main marking process, or main marker for short, traverses data objects from system's root such as the global symbol package list and the active process table. When starting from the active process table, it first marks the data object that represents a process and then scans its stack. The post marking process, or post marker for short, marks data objects that have been notified by write-barriers. Both processes have their own marking stacks. The GC processes are also registered in the active process table, but their stacks will never be scanned, of course.

In the literature, contrary to our approach, sweeping is often embedded in memory allocation primitives in an incremental manner. However, in order not to load symbolic processing primitives with GC overheads, almost all GC jobs are detached from mutator primitives and loaded to the separate sweeping processes that can be very easily interleaved with mutators. This raises, however, some complication in our GC.

4. GC Structure

In this section, we describe the GC structure in some details. Its scheduling will be described in the next Section 5.

With respect to the GC, there are three phases: *GC-off*, *mark*, and *sweep*. We call the *mark* and *sweep* phases *GC-on* phase generically. In a GC-off phase, the GC processes are all sleeping. When one of the six data type categories is detected to be short, then the the GC wakes up into a GC-on phase. Memory shortage is detected by memory allocation primitive such as `cons` and `make-vector`. The remaining amount of free memory is compared with *waterline* of the data type category, that is, a fraction

☆ We use the word "data type category" to denote a generic data type whose basic structure is the same but differently tagged to represent various derivative data types.

of the total memory amount of that data type category, which can be adjusted dynamically by system parameters.

4.1 Mark Phase

When the GC wakes up, the phase is changed from *GC-off* to *mark*, and the main marker and post marker start running concurrently among other mutators.

As was described in the previous section, the main marker begins traversal from the root of the system to find and mark all data objects that are reachable from the root. The most important roots with respect to the real-time GC are the active process table and runnable process queue. The former is a 64 K entry table which represents the set of the current active processes. Each entry is either empty or an active process which is either running, or enqueued in the runnable queue waiting for the CPU resource, or waiting for something that wakes it up to runnable state. Every active process has its own stack except for those which are just to get started. The runnable process queue is a 64-level priority queue, each entry of which is a process queue represented by a (shrinkable) cyclic list; therefore, enqueuing a process may invoke `cons`.

After marking some other small system roots, the main marker scans the active process table sequentially from the top to the bottom. It marks each active process as a data object, then goes to scan its stack. When it finishes marking a process, it advances a pointer to the active process table, called `activeT-front`.

When it finishes scanning the active process table, the *first mark* phase is finished and the *second mark* phase is started. Here, a number of processes may have run during the first mark phase. Under the principle of incremental update, it is enough to scan (maybe again) the stack of the process which has run after the `activeT-front` pointer has passed, or when it stays just at its entry address in the active process table. We call such a process "clobbered" after the Lisp Machine Lisp Manual [23]. Note that even in the second mark phase, newly clobbered processes will appear one after another.

In the second mark phase, the main marker traverses clobbered processes again and again until it catches up with all clobbered processes so that no clobbered processes remain untraversed. Clobbered processes are sought in a 1 K cell bit-table of clobbered processes called `clobBiT`, each bit of which corresponds to an active process table entry. Since scanning of this bit-table may comprise a big inseparable section, there is another 16 cell bit-table called `clobBiTBiT`, which is a bit-table of the former bit-table; if at least one bit of a `clobBiT` 64 bit cell is set, the corresponding bit is set 1 in a `clobBiTBiT` cell. Catching-up is detected by knowing the number of clobbered processes becomes zero. Bits in these bit-tables are set by the micro-kernel in a mark phase, and are cleared by the main marker. Note that the micro-kernel closely cooperates with the GC on these bit-tables.

To make clobbered process marking quick, hardware stack dirty flag is used to decide whether or not a D-block should be scanned again. If the dirty flag of the D-block is not set, it is safe to say that nothing has been changed in the D-block since the last scan, and thereby its scan can be omitted. The two dirty flags for a D-block work independently for different purposes; the other one is used to omit unnecessary swapping from the D-block to the main memory.

4.2 Write-barriers in Mark Phase

In a mark phase, fields of an object already traversed and marked may be changed to reference to another data object instead of what it has been referencing. If nothing special is done here, the newly written reference will not be known by the GC, and the newly referenced object would be collected as garbage. Write-barrier is a well-known mechanism to prevent such accidental reclamation. By virtue of dirty flags, write operation on stack memory does not need a write-barrier.

In the TAO/SILENT microprogram, modification of a field of a data object is done, in principle, by a micro-subroutine call, classified as `wcad` routines. There are about thirty write-barrier subroutines that can fit a variety of conditions about registers and field positions, mainly for the sake of performance optimization. For simplicity, we describe them with a little abstraction, and unless otherwise stated, here we restrict data objects only to `cons` cells.

As was described in Section 2, the mark bit of a cell is the MSB of the car's tag. Hence, which modification, car or cdr, matters. In a mark phase, the modification of cdr, `wcd` is a little simpler than that of car, `wca`, because it need not care about marking bit preservation. The details of `wcad` routines in the mark phase is described in **Fig. 1**, where shallowly markable

```
(wca addr data) =
   (cond ((marked? addr) ; preserve the mark bit
          (write-car-with-mark addr data)
          (shallow-mark data) ) ; do shallow mark
         (:else (write-car-without-mark addr data)) ))
(wcd addr data) =
   (seq (write-cdr-without-mark addr data)
        (shallow-mark data) )
(shallow-mark data) =
    (cond ((no-need-to-mark? data) (nop)) ; immediate data or the like
          ((marked? data) (nop))
          ((shallow-markable? data) (mark data))
          (:else (make-mark-bit-on data)
                 (enter-to-GC-post data) ))
```

Fig. 1 `wcad` routines in the mark phase.

data include those which can be marked quickly, say, within 1 or 2 μsec: 64 bit double floating numbers, bignums, complex numbers, character strings, etc. If an unmarked data object written into a field is not shallowly markable, it is reported to the post marker by posting it to a data buffer called *GC post*. The GC post is simply a stack to which write-barriers push something and from which the post marker pops it off. When the post marker gets CPU, if its own stack is empty, it pops off from the GC post a new data object to be marked, otherwise it continues its own marking with its own stack. Note that the GC post and the post marker's stack are independent.

Posted data is marked before being posted. In the well-known "tricolor" terminology [4], posted data objects can be said colored gray, but it is seen as black from write-barriers, since the mark bit is set. To sum up, there are two marking processes and there is another stack that stores gray objects.

4.3 Sweep Phase

When both markers finish their work ☆, the GC enters a sweep phase, and six sweeping processes, or sweepers for short, start running. The six sweepers correspond to the TAO data type categories: cons cell, vector, buddy, string body, symbol, and binary program blocks called *dytes*. Sweepers are mutually independent and they are scheduled in a round-robin manner, being interleaved with mutators. Each sweeper sleeps as soon as the corresponding data type category is completely swept.

☆ At the last stage of a mark phase, the main marker and the post marker run consecutively to finish the mark phase with a minimum interference by mutators.

Sweeping proceeds from the top to the bottom for every data type category. Each sweeper has a pointer variable representing its sweep frontier. It should be noted that free memory blocks (or *chunks*) can exist ahead of the frontier, because the GC starts when quite a few amount of chunks still remain and some chunks may be concurrently created by explicit deallocation by the system.

4.4 Write-barriers in Sweep Phase

With our concurrent sweepers, write-barriers are needed also in a sweep phase to prevent live objects from being collected as garbage, because modifying a field of a data object may destroy the mark bit information. Moreover, newly allocated data objects should also be prevented from being collected as garbage.☆☆ The latter may not be, in general, called a write-barrier, but we include it here because the purpose is the same.

In a sweep phase, the modification of car, `wca` has to preserve the modified data's mark bit, because if it has been set, the corresponding sweeper does not scan it yet. But the modification of cdr needs no check as shown in **Fig. 2**.

The allocation of a new data object ahead of the corresponding sweep frontier has to set the mark bit, which will be soon erased by the sweeper. For example, `cons` sets the mark bit of a newly cons'ed cell if it is ahead of the cell sweep frontier. Note that `cons` in a mark phase never sets the mark bit, so that very short lived cons cells created in a mark phase are more likely to be collected as garbage in the succeeding sweep phase.

☆☆ Recall that our GC does not do copying or compaction.

```
(wca addr data) =
   (if (marked? addr)
       (write-car-with-mark addr data)
       (write-car-without-mark addr data) )

(wcd addr data) = (write-cdr-without-mark addr data)
```

Fig. 2 `wcad` routines in the sweep phase.

4.5 Bypassing Write-barriers

Some write operations in the system microcode do not need write-barriers. We have nine rules for omitting write-barrier check in order to get rid of the write-barrier overhead. For example, if it is known that a write operation writes only immediate data in the cdr field (or equivalently, writes immediate data into a vector — See Section 4.6), it is sure that no shallow-mark is needed in a mark phase. We listed eight of such typical rules as the microcoding standard. The ninth rule, however, only says "if it can be proved that no write-barrier is needed here, omit it with a proof comment." An example of the ninth rule application can be found in process queue handling in the micro-kernel. Processes of the same priority are enqueued in a (shrinkable) cyclic list of cons cells. When a process is to be dequeued from the queue, an operation like `(setf (cdr x) (cddr x))` is done.☆ This has to be checked by a write-barrier, in general. But one can easily prove that no danger will be incurred by omitting the check. A number of small optimization techniques like this are piled up here and there to minimize the micro-kernel overhead.

4.6 Grinding up the GC

Both markers can be ground up into very fine program segments as can be easily understood, considering parallel marker implementation with a tricolor marking model. Coloring a white object gray, or coloring a gray object black is a simple separate action, and retaining such an invariant that no white object is directly pointed to solely from black object(s) does not involve much computation. Either the first and second mark phases, or stack marking and post marking make no difference with respect to marker grinding.

It is a little harder to grind up the sweepers as fine as the markers, however. The difficulty arises from the facts that variable-size data such as vectors involve a little complicated memory management and that such basic constant-size data as cons cells should be reclaimed as big a chunk as possible for the sake of efficiency, as will be described below.

Variable-size data type categories: vector, buddy, string body, and dytes may be explicitly deallocated by the system microcode if the system knows that the data object has been referenced by only one pointer, which is often the case in TAO/SILENT. Hence, these sweepers simply share the code of explicit deallocation. This implies that each time a data object of these types is collected as garbage, everything is neat with respect to the memory management status. Hence, it is a simple matter to grind up these sweepers to some fine grain. However, because it needs some bounded amount of computation to merge newly reclaimed data into adjacent free chunk(s), it is difficult to grind them up equal to the granularity of the markers. Free chunks of these variable-size data type categories are classified according to their sizes, in a table whose entries correspond to roughly 2's power sizes so that allocation does not involve search in the chunk chain.☆☆

For cells, it is not as simple as could be imagined. A simple-minded implementation would collect each unused cell and link it to free cell chain one by one. In that case, it is quite simple to grind up the cell sweeper. However, this simple-minded implementation suffers from poor performance, because it invokes memory write operation for every reclaimed cell.☆☆☆

Our cell sweeper, instead, collects contiguous free cells as one free chunk as long as the sweeper is not preempted by the sched-

☆ We borrow the expression from Common Lisp for explanation convenience. In TAO, it is written as `(!(cdr x) (cddr x))` or `(!!cdr !(cdr x))`.

☆☆ It is also easy to grind up the symbol sweeper. TAO collects symbols as garbage if they are not likely to be referenced any more. For the symbol GC, each symbol has two marking bits, one of which is set only via its symbol package. But we do not go into further details, because it is not much relevant to a real-time GC.

☆☆☆ Write operation takes more machine cycles than read operation in SILENT.

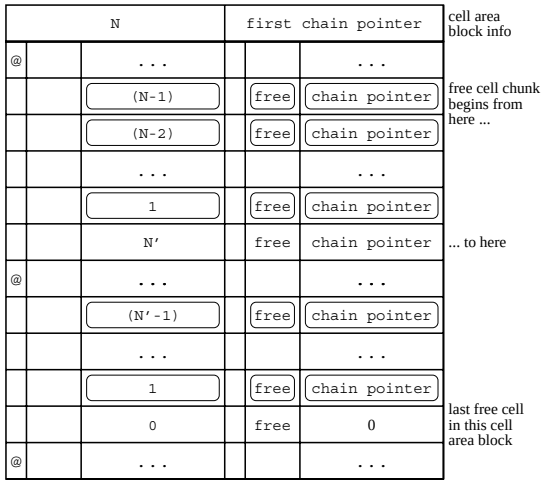

Fig. 3 Internal structure of a free cell area. A cell area is a 32 K cell memory block. Its header contains the link to the first free chunk below. All but the last cell need not be initialized since a register that points to the first free cell serves as a cache. Of course, this is trivially realized in a copying or compactifying GC.

uler. Only one write operation is performed per chunk in a time slice (See **Fig. 3**). For every live cell, however, the sweeper must clear its mark bit. An experiment shows that if the cell sweeper writes something in every reclaimed cell, it slows down by a factor of two.

Free chunks that have been existing before the GC starts can be easily merged with those newly reclaimed. From the allocator's view point, the free cell pointer into a chunk can be cached in a register, and typically, `cons` has only to advance the free cell pointer and decrease the remaining size of the chunk, which is also cached in another register, without referring to the contents of free cells unless it reaches the last cell of the chunk. This is the "no write as far as possible" principle prevailing in the TAO/SILENT implementation.

This straightforward improvement brought in some complexity in the logic of cell sweeper grinding, because the cell sweep frontier and free cell pointer can arbitrarily interfere each other by racing caused by interleaving. This was the most complicated and hard-to-debug part in our GC. The algorithm and its correctness proof are big enough and deserve to be explained in an independent paper.

4.7 No Write as far as Possible

The "no write as far as possible" principle is applied everywhere in our GC. Typical one in the cell sweeper was described in the previous section.

Each vector has only one mark bit at its head's car. The cdr of the vector head contains the vector size. Writing into a vector element need not care about mark bit. That is, vector element assignment needs only `wcd` type write-barrier that is lighter than `wca`. It also makes the GC possible to omit reading or writing mark bits in vector elements. By virtue of the vector size description in the head's cdr, the vector sweeper can skip lightly from one vector (or chunk) head to a vector (or chunk) head immediately below.

Each buddy has only two mark bits at its head cell's car and its last cell's car. Buddy has a size of 2's power and is used for various system internal data types. Pointer to a buddy contains its size information embedded in the tag or the address part so that no header information is needed to be stored in the buddy itself. This saving gives an advantage of lighter write-barriers. On the contrary, the buddy sweeper has to know the size of live buddy by examining the estimated last car's mark bit by doubling the size estimation every time until it meets the answer, i.e., corresponding marked last cell's car.

To make variable-size data sweepers fast, the contents of variable-size data are not cleared when it is reclaimed, except for some minimal administrative information at chunk's boundary such as forward/backward free chain pointers. Hence, chunks may contain obsolete data references inside. When they are to be reused by later allocation, these useless and dangerous references should be hidden from the marker's accidental traversal. Such an accident can, of course, be simply circumvented by allocation time initialization. But the initialization would take arbitrarily long time if, say, a newly allocated vector is very large. Hence, a big vector or buddy can have an "under initialization" state so that the markers do not go further into the vector or buddy elements. Thus, large data block initialization with no non-referencing data can be safely interleaved with other processes.

4.8 Hardware Support and Coding Techniques

SILENT has no very special hardware feature devoted to the GC except for the hardware dirty flags described in Section 4.1. Some of its symbolic processing architecture, however, contribute to enhance the GC performance.

Write-barriers are implemented as a micro-subroutine which is called, say, by the following microinstruction form:☆

(<ALU operation ... > (bsr gcm wca))

where bsr denotes multiple entry subroutine call to wca. One of the three entries of a wca routine will be selected by the condition gcm, which is an alias of psw3-2, which means bit 3 and bit 2 of PSW (processor status word). The SILENT PSW has four user definable bits (bit 3-0). Our GC uses bit 3-2 to represent one of the three phases of the GC, and uses bit 1 to distinguish the GC-off phase from the GC-on phase. This implies that no explicit check action beforehand is needed to know in which phase it is at present. If the gcm condition is used with a simple multi-way branch instructions br and bsr, the phase check is completely transparent (i.e., overhead-less) when no special action for the GC is needed.

Multi-way branch instructions are more or less essential to implement a dynamic language like TAO. Our GC enjoys its functionality extensively to dispatch the marking disposal according to about sixty data types in TAO. In the microcode of the markers (including the shallow marker), 64-way branch case statements appear at seven places for efficiency, which are very similar to each other but definitely different in some portion. This is a sort of "inline code" optimization.

With the aid of such inline coding, the innermost loop for stack scan consists of only two dynamic steps if it finds no-need-to-mark object in a stack word. Hence, there is no room to check hap-interrupts (described soon later) in the loop. In a rare case in which there is no data object reference in a 128 word D-block, 256 micro-steps (7.7 μsec) will run without being interleaved; this is one of the longest inseparable sections in our GC.

There are a number of other small coding techniques around our GC. For example, to grind up the cell sweeper thoroughly, the main sweep loop, rather big one, is duplicated with alternating register assignments, because if only one set of register assignment is used, register value update needs superfluous one micro-cycles. By this loop duplication, the (shortest) interrupt check interval is shortened by one micro-step to 4 micro-steps.

☆ As can be seen, microprogram itself is written in the S-expression

5. GC Scheduling

The scheduling policy is one of the most crucial points in a concurrent real-time GC. In this section, we describe how the GC is invoked, and how GC processes are scheduled among mutators.

Eight GC processes are scheduled slightly differently from mutators. Mutators are given the CPU time by a priority-based preemptive scheduling with round-robin rotation. Each process in the same priority queue is given an 8 millisecond quota. On the other hand, the GC processes are not enqueued in the priority queues, and they are scheduled by the degree of memory shortage (called *GC urgency*), instead. The priority of all GC processes is set to 4, an apparently lower priority which does not hinder important process scheduling even when it is referred for priority comparison.

5.1 General Flow of Interrupt Disposal

Before going further, we had better introduce here some peculiar points on the SILENT interrupts. SILENT has two level interrupts: micro-interrupt and hap-interrupt. The former is a usual low-level hardware interrupt that takes over the control from currently executed microinstruction. Maskable micro-interrupts are caused typically by timer overflow and external device requests. As can be easily understood, micro-interrupts cannot directly invoke the process scheduler, because they can happen anywhere in the system's inseparable primitive microcode.

The hap-interrupt is not a genuine hardware interrupt, where hap stands for "happen." It is notified to the system by setting the corresponding PSW bit and detected by an explicit hap check in a microinstruction, which can be done in parallel with usual arithmetic micro-operation. The causes of hap-interrupts are stack overflow, machine-cycle counter down-to-zero, special communication channel requests, and user raised hap-event. The last one is called simhap (stands for "simulated hap"), which can be set by a special instruction field in a microinstruction.

Thus, an external event first causes a micro-interrupt, then its disposal routine sets simhap condition if necessary, and eventually some hap check detects the event at a timing in which there remains no dangling pointer or halfway computation. If a hap urges process scheduling,

then the micro-kernel selects the next process to run and, if necessary, swaps its stack into the hardware stack memory before passing the control to the process.☆

The time between the first micro-interrupt acceptance and the first microinstruction execution of the process waken up by the interrupt is the first half of the response delay, i.e., wake-up delay defined in Section 1.

5.2 GC Invocation

The GC processes first get started at bootstrapping and immediately go to sleep. When the free memory of any one of six data type categories is detected to become short by the corresponding allocation routine such as `cons` and `make-vector`, a small subroutine is called to set `simhap` condition for invoking the GC. Soon, a `hap` check detects the `simhap` and calls the scheduler. This indirection for calling the scheduler is necessary since the allocation routine mostly detects the memory shortage deep in an inseparable section.

Checking the memory shortage itself is an overhead to the allocation routine, however; it is not negligible if the routine is very light as `cons` which takes typically only 5 micro-cycles. Hence, `cons` does the check only when the free cell pointer is to go across the boundary of a 32 K cell memory block. That is, the check is done with the probability far below one thousandth; thereby the overhead is negligible. Other data type categories have bigger allocation routines so that the waterline checking overhead is buried in the allocation bureaucracy.

It is an important issue how high each waterline should be set to detect the memory shortage. If memory consumption rate is high, waterline should be also high for the GC to catch up with mutators as will be show in Section 6.3.

5.3 GC Urgency

In a GC-on phase, GC processes are scheduled according to parameters representing the urgency of the GC. The GC urgency is classified into 4 degrees depending on the amount of remaining free memory. Roughly speaking, for the lowest urgency, GC processes run at every 9th scheduling, but for the highest urgency, GC processes run at every second scheduling; that is, GC processes and mutators run alternatively, unless urgent real-time processes of priority more than 47 are waiting for the CPU time. Defaulted quota given to the GC processes is 15.5 milliseconds. Thus, in most urgent situation, the GC get twice as much CPU time as mutators. In a GC-on phase, the GC urgency is updated every time a GC process is about to run.

Table 1 GC urgency.

urgency	criterion	GC frequency
0	$1 > F > 1/2$	every 9th
1	$1/2 \geq F > 1/4$	every 5th
2	$1/4 \geq F > 1/8$	every 3rd
3	$1/8 \geq F$	every 2nd

The rule about the GC urgency, which was determined empirically, is shown in **Table 1**, where F means the ratio of two measures of the data category whose shortage has invoked the GC: the amount of available free memory and the waterline. The GC frequency denotes an approximate scheduling chance of GC processes among mutators.

In this setting, we observed that the busiest GC scheduling occurs only a few percents of runs of GC processes around the tansition from mark phase to sweep phase in a GC invocation, even in severest tests. Some of concrete data will be presented below.

6. Evaluation

We evaluated the GC performance in two ways: counting the overhead micro-cycles induced by this concurrent GC, and measuring the time (in fact, counting micro-cycles) of benchmark program execution.☆☆

6.1 Overhead Micro-cycles

Dynamic steps for write-barriers are the main overhead brought in to Lisp primitives in our implementation. **Table 2** shows the number of machine-cycles for write-barriers that are not needed in a stop mark-sweep GC.
where a range m–n means that m is the minimum overhead and n is the maximum overhead, i.e., the overhead for the worst case in which every branch goes to longer code path and every memory access misses the cache. Writing into an object slot and vector element has the same overhead as rewriting cdr as was described in

☆ During swapping-in, another process of higher priority may preempt the process, of course.

☆☆ SILENT has a micro-cycle counter and a number of other statistics counters of a micro-cycle resolution. Fluctuation of the time measurement arises from the asynchronicity of external interrupts and periodical DRAM refreshing that delays main memory access 3 or 4 cycles.

Table 2 Overhead for write-barriers.

	GC-off	mark	sweep
`cons`	0	0	1–2
rewriting car	1	5–37	5–11
rewriting cdr	1	3–26	1

```
(defun fib (n)
  (if (<= n 1)
    1
    (+ (fib (1- n)) (fib (- n 2))) ))
```

Fig. 4 Fibonacci function.

Section 4.5.

There is no overhead in access or assignment to a local variable, function call, or such read access primitives as `car`, `cdr`, and `vector-ref`.

As was described before, waterline check overheads are negligible.

Only a small overhead is incurred in the process management of the micro-kernel. For example, in a GC-on phase, additional four machine-cycles are needed to make a runnable mutator run because of the GC urgency check. However, setting the bit-tables for a clobbered process takes 40 to 50 machine-cycles every time a clobbered process releases the CPU; this is the worst overhead incurred in the micro-kernel.

6.2 Benchmark Programs

We used the following two types of scalable benchmark programs to evaluate the actual GC performance: `mfib-loop` and `cell-eater`.

(1) `mfib-loop`

This is a benchmark written in TAO mainly for measuring wake-up delay in most severe conditions (See Appendix A.8). It looks a little complicated, but it is a simple extension of recursive definition for Fibonacci function in **Fig. 4**.

Every recursive call is replaced by a process spawning and two returned values are passed from the spawned processes by a mailbox, which is a primitive data type for blocking interprocess communication with an unbounded buffer. This simple extension does not consume cells so much other than those for runnable process queue and mailboxes. So a simple trick is embedded to waste more cells and enlarge the marking root.

(2) `cell-eater`

This is a benchmark to measure the GC speed rather than its response time (See Appendix A.9). `cell-eater` calculates the sum of integers from 1 to n. There is only one mutator which wastes cells at its full speed. However, we imposed it a reasonable condition that it must access at least once to every cell it has cons'ed. The more are *live* cells, the more difficult is the GC to catch up with the mutator because a mark phase takes more time and the mutator takes less time to use up remaining free cells. Hence, the cell waterline should be raised if the number of live cells is raised. We measured the marginal percentage of live cells for given cell waterline.

Table 3 Result of `(mfib-loop 17 200 120 70)`.

	time [μsec]	percentage
elapsed time	1,149,427,948	100.00
net computation	556,703,988	48.43
micro-kernel (1)	149,980,058	13.05
GC processes (2)	442,743,902	38.52
main marker (3)	344,893,464	30.00
post marker (4)	5,973,642	0.52
cell sweeper	78,826,828	6.86
vector sweeper	2,038,127	0.18
buddy sweeper	8,242,238	0.72

(1) management of processes and stacks
(2) sum of all GC processes' CPU time
(3) big root in this benchmark
(4) small in this benchmark

6.3 GC Speed

We first show the GC speed for both benchmarks. We did a lot of experiments with varying parameters, but the performance figures can be well represented by the following results.

(1) `(mfib-loop 17 200 120 70)`

This benchmark holds a big marking root (processes and their stacks), but does not use so much cells (**Table 3**). Here we set the size of cell area 544 K cells to make the time for GC-off phases and GC-on phases even. In this setting, one GC invocation has to scan as much as 1.6 M words in mutator stacks, 95% of which are swapped out into the main memory when they are scanned. Stack dirty flags save 38% of D-block scanning. The number of GC invocations is 753 and the total number of spawned processes are 1,038,363. Here, figures for some sweepers are omitted.

In this somewhat GC burdening benchmark, each GC invocation behaves quite differently. Some need only less than 40 runs of GC processes with the lowest GC urgency, but a few others (below 5 percents of the entire GC's) need more than 150 runs of GC processes with three to five out of them running in the high-

Table 4 Result of `(cell-eater 1000000 1000 live)`.

	time [μsec]	percentage
elapsed time	795,993,783	100.00
net computation	604,767,619	75.98
micro-kernel (1)	1,859,869	0.23
GC processes (2)	189,366,296	23.79
main marker (3)	2,023,024	0.25
post marker (4)	107	0.00
cell sweeper (5)	186,869,769	23.48
vector sweeper	30,655	0.00
buddy sweeper	18,422	0.00

(1) nearly zero, of course
(2) about 1/4 of elapsed time
(3) little
(4) further little
(5) speed of cell reclamation

est GC urgency. The curve of GC urgency history in a GC invocation takes a gradual mountain-like shape whose summit is located at the boundary of the mark phase and the sweep phase where the GC and mutators compete most critically. A typical GC urgency history is, say, 0(7), 1(30), 2(55) in mark phase and 2(8), 1(1), 0(4) in sweep phase where $m(n)$ means urgency m runs n times in a row. Scheduling based on the GC urgency certainly resolved the well-known problem of the transition from mark phase to sweep phase in a concurrent GC.

For bigger n, say 22, the GC still catches up with those mutators, where the maximum number of concurrent processes is 34,817. We once ran concurrently a set of this benchmarks with varying parameters, giving bigger loop count for each benchmark. The program ran over 100 days without trouble.

(2) `(cell-eater 1000000 1000` *live*`)`

For *live* = 0 and 8 M free cell area, we have the result shown in **Table 4**. The number of GC invocations is 116, and the number of spawned process is 1. It shows that the speed of cell reclamation is about 3 times as much as cell consumption in this simple setting.

Figure 5 shows the marginal percentage of live cells for given cell waterline (in percentage). As can be seen, even in this pathological situation, our concurrent GC can tolerate nearly 50% of live cells.

6.4 Wake-up Delay

Response delay is measured by running concurrently a trivial real-time process which only receives the interrupt signal with the time stamp of its arrival to SILENT, and calculates the time difference from the current time, i.e., wake-up delay in our terminology. In this

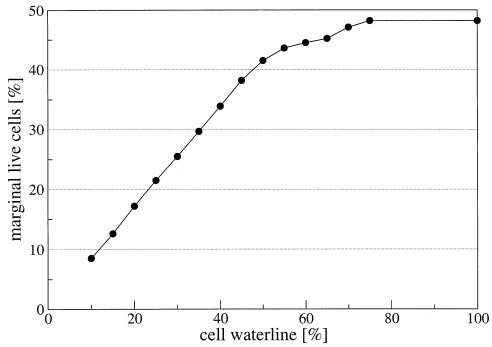

Fig. 5 Marginal live cell percentage in `cell-eater`.

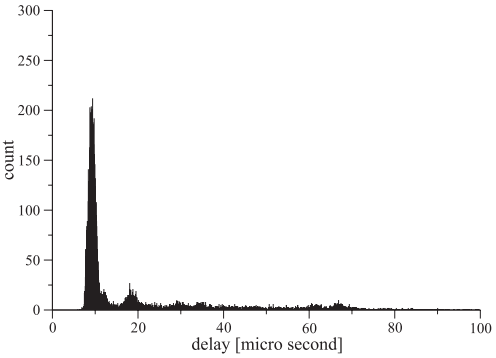

Fig. 6 Distribution of wake-up delays in GC-off phase.

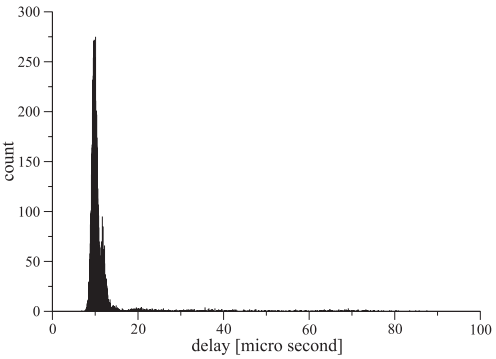

Fig. 7 Distribution of wake-up delays in GC-on phase.

measurement, external events are raised every 33.3 milliseconds from the front-end processor of SILENT, having animation graphics in mind.

(1) `(mfib-loop 17 200 120 70)`

Totally, 33,963 interrupts take place. In the setting described in Section 6.3, about a half of them take place in GC-off phases, and the other half in GC-on phases. **Figures 6** and **7** show the distribution of wake-up delays.

Table 5 summarizes the resulted delay in

Table 5 Resulted delay for `mfib-loop`.

	samples	average delay	σ	worst delay
total	33,963	14.998	13.890	119.070
phase				
GC-on	15,944	11.280	6.393	87.510
GC-off	18,019	18.289	17.447	119.070

Table 6 Resulted delay for `cell-eater`.

	samples	average delay	σ	worst delay
total	23,590	11.039	1.319	19.320

μsec, where σ denotes standard deviation.

Contrary to most people's intuition (but so we anticipated), the average wake-up delay is smaller in GC-on phases than in GC-off phases. The reason is obvious; GC grinding-up achieves much finer grain than the micro-kernel, which contains at worst 70 μsec inseparable stack swapping.

However, the worst wake-up delay does not depend on the GC phases. Several all day long measurements with finer logs reported that the worst measured wake-up time was 130.3 μsec. We found that in that case there is a `hap`-interrupt check 8 steps before. So we can more or less safely estimate the worst wake-up delay is around 131 μsec for this benchmark, which is the severest one conceivable with respect to wake-up delay.

(2) `(cell-eater 100000 1000 0)`

As can be easily imagined, this test brings in much better result with respect to wake-up delay, because process switching time is best possible in this benchmark. If the user wants a truly real-time application on TAO/SILENT, he/she surely designs the system configuration not so far different from this. In **Table 6** (wake-up delay in μsec), we do not discriminate GC-off and GC-on phases, since they make little difference.

The reason for this small delay is that this benchmark does not contain lengthy inseparable sections. Our principle to microcode TAO primitives is "do not write inseparable section that needs more than 30 μsec." If this is strictly obeyed, about 50 μsec wake-up delay would be guaranteed in practical situations.

6.5 GC-post Capacity

The post marker accepts "gray objects" posted by write-barriers. It is important to make sure that the GC-post will not overflow even in the worst case. The following program is conceivably a most malicious one:

```
(defun write-barrier-teaser (count)
  (let ((x (list 0)))
     (dotimes count (!(car x) (list 1))
        (!(car x) (list 2))
        (!(car x) (list 3)) )))
```

With 8 M cells and `count`=100,000,000, the observed maximum usage of the GC-post was about 40,000 words, which is smaller than the intuitively designed GC-post size, 64 K words. We can say the GC-post is tolerable against this malicious write-barrier teaser. It is, of course, possible to make further worse setting against the GC-post, but we may not care about such pathological cases.

7. Concluding Remarks

In the last of Section 1, we said that our GC is *transparent* to real-time processes. Now, it is clear why we could say so. If real-time processes do small urgent work without consuming much amount of resources, its wake-up delay is the main concern, since in our GC, even in a GC-on phase, Lisp primitives such as `car` and `cons` do not suffer from GC overheads. We showed that major wake-up delays may not be derived from GC but from the micro-kernel and lengthy Lisp primitives. In other words, real-time processes will seldom be aware of the GC; they can see only the overheads of the micro-kernel and Lisp itself, however. This is exactly what we mean by the phrase "our GC is transparent."

The wake-up delay of the figures, 50–160 μsec, not milliseconds, turned out to be more than order of magnitudes smaller than other related researches, known up to 1999 except for those done by Roger Henriksson [7], when we finished the implementation. Note that the machine on which we implemented the GC is considerably slower than the currently available processors; it's clock is only 33 MHz. We wonder why this simple idea based upon incremental update with write-barriers has not been fully and thoroughly realized so long. Nothing very special and novel is involved in our basic algorithm and implementation.

Here we have to mention Henriksson's outstanding work which achieved a comparable wake-up delay with our GC. His real-time GC is, as he called, a semi-concurrent GC, in which the incremental GC work is interleaved only with low-priority processes, and gets be-

Appendix

```
(defun mfib-loop (n k i j)
    (let ((v (mfib n i j)) (c 0))
      ; The meaning of :until and :while may be obvious.
      (loop (:until (= c k) 'congratulation)
            (:while (= v (mfib n i j)) c) ; make sure the result is correct.
            ; "(!" ≈ "(setf"
            (!c (1+ c)) )))
(defun mfib (n i j)
    (if (<= n 1) 1
        ; (make-process 20 0) for priority 20, protection level 0.
        (let ((p1 (make-process 20 0))
              (p2 (make-process 20 0))
              ; The following line ≈ multiple-value-bind of Common Lisp.
              ; make-mailbox returns two values: the output port and input port.
              ; Two dots .. designate that two values will be received.
              ; mbo for output port and mbi for input port.
              ((mbo mbi) ..(make-mailbox)) )
          ; (make-chore fn args) makes a chore: pair of a function and its arguments.
          ; (task proc chore) gives a process a chore to be executed.
          (task p1 (make-chore #'mfib+ (list i mbo (1- n) i j)))
          (task p2 (make-chore #'mfib+ (list j mbo (- n 2) i j)))
          (+ (receive-mail mbi) (receive-mail mbi)) )))
; check the sum
(defun mfib+ (m mb n i j)
    (if (= (sigma-2 (mfib+sub m mb n i j)) (sigma m)) #t (error)) )
; call mfib when the process's stack grows at maximum — make the marking root bigger,
; nevertheless, returned value is the series sum.
(defun mfib+sub (m mb n i j)
    (if (= m 0)
        (block (send-mail mb (mfib n i j)) (list (list 0)))
        (cons (list m) (mfib+sub (1- m) mb n i j)) ))
; get the sum of series embedded in a cell wasting list.
(defun sigma-2 (list) (if (empty? list) 0 (+ (caar list) (sigma-2 (cdr list)))))
; normal series summation for reference
(defun sigma (n) (if (= n 0) 0 (+ n (sigma (1- n)))))
```

Fig. 8 A process spawning Fibonacci function.

```
(defun cell-eater (p q r)
    ; (make-deep-list r) makes a deep list structure consisting of exactly r cells.
    (let ((live-list (make-deep-list r)) (c 0) (k (sigma q)))
      (loop (:until (= c p) #t)
            (:while (= k (sigma-list (list-n q))) (error))
            (!c (1+ c)) )))
(defun list-n (n)
    (if (= n 0) (list (list (list (list (list (list (list (list (list 0))))))))
        (cons (list (list (list (list (list (list (list n)))))))
              (list-n (1- n)) )))
(defun sigma-list (lst)
    (if (empty? lst) 0 (+ (caaaar (caaaar lst)) (sigma-list (cdr lst)))) )
```

Fig. 9 Cell eater.

hind when a limited number of high-priority processes run in order to assure quick response for them. That is, low-priority processes and high-priority real-time processes are dealt separately with respect to memory allocation. Henriksson's approach is akin to ours in pursuing a minimal response delay, but different in the degree of concurrency. Our GC is very conscious of the GC performance as well as the response time as could be seen in our overloading experiments, while his prototype implementation and experiments were not.

The following can be said as the reasons of the remarkable performance of our GC:

(1) Inseparable sections can be easily ground up by virtue of two stage interrupt handling.
(2) The GC and the operating system, namely the micro-kernel, are cooperating so intimately that the GC can have its own scheduling policy and enjoy the light-weight process mechanism.
(3) Our GC is simple; it does not employ compaction.
(4) A number of tiny but non-trivial programming techniques are piled up.
(5) A few hardware features assist the implementation.

Among all, we would like to emphasize the effectiveness of the cooperation of the GC and the operating system. The GC is another kind of *virtual memory* mechanisms, which provides an inexhaustive fountain of recycled memory though there is only a finite amount of memory in a machine. In other words, the GC can be said a time-axis virtual memory, while the usual one is space-axis virtual memory; the latter has long been one of the main issues of operating systems research. Why not for GC? We insist that the GC, especially real-time GC, should be one of the most important topics in the operating systems research.

Some of the above-mentioned reasons and remarks may not be directly applicable to other GC implementations on the current stock machines. Considering the astonishingly growing speed of micro-processors, however, we believe that the techniques of this simple-minded but a little tough-to-implement GC can be applied to future symbolic processing systems with much higher performance figures.

References

1) Appel, A.W., Ellis, J.R. and Li, K.: Real-time Concurrent Collection on Stock Multiprocessors, *ACM SIGPLAN Notices*, Vol.23, No.7, pp.11–23 (1988).
2) Henry, G. and Baker, J.: List processing in real time on a serial computer, *Comm. ACM*, Vol.21, No.4, pp.66–70 (1978).
3) Boehm, H.-J., Demers, A.J. and Shenker, S.: Mostly parallel garbage collection, *ACM SIGPLAN Notices*, Vol.26, No.6, pp.157–164 (1991).
4) Dijkstra, E.W., Lamport, L., Martin, A., Scholten, C. and Steffens, E.: On-the-fly garbage collection: An exercise in Cooperation, *Comm. ACM*, Vol.21, No.11, pp.966–975 (1978).
5) Henriksson, R.: Scheduling real time garbage collection, Technical Report LU-CS-TR: 94-129, Department of Computer Science, Lund University (1994).
6) Seligmann, J. and Grarup, S.: Incremental Mature Garbage Collection Using the Train Algorithm, *Proc. ECOOP'95, Ninth European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Vol.952, pp.235–252 Springer-Verlag, New York, NY, USA (1995).
7) Henriksson, R.: Scheduling Garbage Collection in Embedded Systems, Technical Report LUTEDX/(TECS-1008)/1-164/(1998), Department of Computer Science, Lund Institute of Technology, Lund University (1998).
8) Huelsbergen, L. and Winterbottom, P.: Very concurrent mark-&-sweep garbage collection without fine-grain synchronization, *Proc. First International Symposium on Memory Management*, pp.166–175, ACM (1998).
9) Johnstone, M.S.: *Non-Compacting Memory Allocation and Real-Time Garbage Collection*, Ph.D. Thesis, University of Texas (1997).
10) Jones, R. and Lins, R.: *Garbage Collection*, John Wiley & Sons (1996).
11) Kung, H. and Song, S.: An efficient parallel garbage collection system and its correctness proof, *Proc. Eighteenth Annual Symposium on Foundations of Computer Science*, IEEE, Providence, Rhode Island, USA, IEEE, pp.120–131 New York, USA (1977).
12) Lim, T.F., Pardyak, P. and Bershad, B.N.: A memory-efficient real-time non-copying garbage-collector, *Proc. First International Symposium on Memory Management*, pp.118–129, ACM Press, Vancouver (1998).
13) Nilson, K.D. and Schmidt, W.J.: Hardware-Assisted General-Purpose Garbage Collection

for Hard Real-Time Systems, Technical Report ISU TR 92-15, Iowa State University (1992).
14) Yuasa, T.: Real-time garbage collection on general-purpose machines, *Journal of Systems and Software*, Vol.11, pp.181–198 (1990).
15) Robertz, S.G.: Applying priorities to memory allocation, *ACM SIGPLAN Notices*, Vol.38, No.2s, pp.108–118 (2003).
16) Endo, T. and Taura, K.: Reducing pause time of conservative collectors, *ACM SIGPLAN Notices*, Vol.38, No.2s, pp.119–131 (2003).
17) Stankovic, J.: Real-Time Computing Systems — The Next Generation, *IEEE Tutorial on Hard Real-Time Systems*, Stankovic, J. and Ramamritham, K. (eds.), pp.14–37, IEEE (1988).
18) Hibino, Y., Watanabe, K. and Takeuchi, I.: A 32-bit LISP Processor for the AI Workstation ELIS with a Multiple Programming Paradigm Language TAO, *Journal of Information Processing*, Vol.13, No.2, pp.156–164 (1990).
19) Yamazaki, K., Amagai, Y., Takeuchi, I. and Yoshida, M.: TAO: an object orientation kernel, *Proc. First JSSST International Symposium on Object Technologies for Advanced Software*, Nishio, S. and Yonezawa, A. (eds.), Lecture Notes in Computer Science, Vol.742, pp.61–76 New York, NY, USA, Springer-Verlag (1993).
20) Takeuchi, I., Okuno, H.G. and Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms, *New Generation Computing*, Vol.4, No.4, pp.401–444 (1986).
21) Takeuchi, I., Amagai, Y., Yamazaki, K. and Yoshida, M.: Lightweight processes in the real-time symbolic processing system TAO/SILENT, *Advanced Lisp Technology*, Yuasa, T. and Okuno, H. (eds.), pp.135–154 IPSJ, Taylor & Francis (2002).
22) Wilson, P.R.: Uniprocessor garbage collection techniques, Technical report, University of Texas (1994). Expanded version of the IWMM94.
23) Weinreb, D., Moon, D. and R.M.S.: *Lisp Machine Manual*, fifth edition system version 92 edition (1983).

(Received May 20, 2003)
(Accepted July 8, 2003)

Ikuo Takeuchi received his B.S. and M.S. degrees in mathematics in 1969 and 1971, respectively, and Ph.D. of engineering in 1996, all from The University of Tokyo. He had been working for Nippon Telegraph and Telephone Corporation since 1971 till 1997, and now is a professor of the University of Electro-Communications.

Yoshiji Amagai received his B.E. and M.E. degrees in 1983, 1985 in computer science and communication engineering from the University of Electro-Communications. He has been in Nippon Telegraph and Telephone Corporation from 1985, and is currently working at NTT Network Innovation Laboratories.

Masaharu Yoshida received his B.E. degree in electrical engineering in 1976 and his M.E. degree in electrical engineering in 1978, all from Chiba University. He has been in Nippon Telegraph and Telephone Corporation from 1978, and is currently working at NTT-IT Corporation.

Kenichi Yamazaki received his B.E. and M.E. degrees in 1984 and 1986 respectively, all from Tohoku University, and his Ph.D. degree in 2001 from the University of Electro-Communications. He had been working for Nippon Telegraph and Telephone Corporation since 1986, and is currently working for NTT DoCoMo Incorporation, Network Laboratories.