Lisp can be "Hard" Real Time

Ikuo Takeuchi (The University of Electro-Communications),
Kenichi Yamazaki, Yoshiji Amagai, Masaharu Yoshida (NTT)
nue, yamazaki, amagai, hal@nue.org

1 Introduction

Lisp has long been believed not suitable for hard real-time applications. It is still true if the term "hard real-time" is used in the most rigorous meaning. However, there have been developed a variety of techniques to adopt Lisp for a certain level of real-time applications. At least, Lisp is now being used for a number of soft real time applications.

For being real-time, a system has to be responsible to external events within a specified limit of time, and it has also to finish given jobs within specified deadlines. If any violation of deadline causes fatal system collapse, the system is not called hard real-time; however, nothing fatal happens at infrequent deadline violation, the system is called soft real-time.

For Lisp, there are two major obstacles to make it suitable for real-time applications. The first is long pause time caused by the garbage collection (or GC hereafter). The second is the unpredictability of resource usage including the CPU time. Lisp programmers cannot rely on a predefined scheduling scheme. In the first place, people says that Lisp is a language for artificial intelligence, and intelligence is not the matter of hard real-time nature.

However, growing number of applications need a certain amount of intelligence while meeting severe real-time requirement. So it is still a challenge to make Lisp more suitable for, hopefully, hard real-time.

Reflecting the two major obstacles, our approach to achieve this goal is characterized by the following two ideas:

(1) Make Lisp be able to respond more quickly to external events even while GC is running concurrently with mutator processes. That is, minimize pause time caused by GC; in addition, minimize the inseparable sequence of code (or inseparable section) for any Lisp primitive to allow external interrupt be accepted as soon as possible.

(2) Clarify and guarantee the execution time of every Lisp primitive as a function of its argument complexity. For example, if the user wants to `nconc` a list consisting of n top-level cells with another list, the user can know its worst case computation time as a function of n from the language reference manual.

A real-time symbolic processing system on a single processor should have the following four features:

- powerful multiprogramming capability,
- interrupt handling facilities that can handle external events and internal clocks etc.,
- no long inseparable sections inside the system, and finally but not least,
- real-time GC.

It should be noted that pause time caused by GC concerns only the first half of the response delay, and the computation performed by a real-time responsive process, (or simply, 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* in the paper. We do not describe much about the second half here. If real-time processes do only a fixed amount of simple computation every time, the system can become hard real-time; but if Lisp is used, the user may expect more or less flexible computation, which implies at most soft real-time by definition.[1]

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

It is at least, however, important to guarantee better performance figures of symbolic processing primitives, with which the user can deliberately design his/her real-time application profile. That is, if a real-time GC imposes much overheads on these primitives, a real-time process suffers from poorer performance for its actual computation even if it is waken up very quickly by an urgent event.

There may tend to 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 fractions as fine as possible, while 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 such inseparable sections should be restricted by the runtime system parameters for severe real-time applications.

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 mutators that consume and modify data objects. In other words, GC should be incremental because it is impossible to finish all the collection at once. In addition to being quickly responsive, it should have a reasonable speed of collecting unused (unreferenced) data objects so that it can always supply enough fresh data objects to mutators.

In this paper, we describe the GC implemented by microprogram on the real-time symbolic processing system TAO/SILENT which is being developed by us. As we explained later, this GC is in a sense almost *transparent* to real-time processes. The performance we achieved here will encourage Lisp lovers to be sure Lisp can be used for "hard" real-time applications.

2 TAO/SILENT

Before describing our GC, we sketch the TAO/SILENT system briefly.

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 [Hibi90]. TAO 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 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.

Assumed applications of TAO/SILENT include real-time object-oriented computer animation, intelligent network control and real-world robotics.

SILENT is a micro-programmable 40 bit word machine; 8 for tag, and 32 for pointer or immediate data. The MSB (most significant bit) of the car's 8 bit tag is used for GC mark bit.

SILENT is now equipped with a 256K word hardware stack memory whose push or pop can be done in one machine cycle. 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 33MHz, which is not very fast compared with the current microprocessors. 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 (micro-kernel) is written fully in 80 bits/word horizontal microcode. It can control up to 64K 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 [Take00].

Processes can have either one of 64 priority and four protection levels, where bigger number corresponds to higher priority and protection level, respectively. Micro-kernel employs a preemptive priority-based scheduling with round-robin.

One of the important jobs of the micro-kernel is 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. It occasionally transfers a process's stack to a different location in the hardware stack memory since SILENT does not have MMU (memory management unit).

For the stack management, we group each four contiguous D-blocks 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 256K words, all of them can coexist on the hardware stack memory and enjoy fastest process switching, say, 4 μsec.

There are, however, a few long inseparable sections in the micro-kernel, mainly for the stack swapping and/or transfer mentioned above. 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, but it uses at most less than 500 stack words, which are enough for a few dozens of nested function calls. That is, real-time process itself will not load the micro-kernel down so much. We believe that this assumption is reasonable.

SILENT has a micro-cycle counter and a number of other statistics counters with the micro-cycle resolution. Hence, we can measure the time of benchmark tests with 30 nanosecond resolution, and we can count up an exact number of executions of a designated set of microinstructions. Fluctuation of the time measurement arises from the asynchronicity of external interrupts and periodical DRAM refreshing which delays main memory access 3 or 4 cycles.

3 Basic GC strategy

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

- incremental update with write-barrier,
- mark-sweep,
- no compaction, and
- very concurrent marking and sweeping processes.

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

As was depicted in [Wils94], incremental update write-barrier algorithm is in various aspects better than snapshot-at-the-beginning write-barrier algorithm [Yuas90]. For example, it is less conservative, i.e., more garbages are likely to be collected, thereby the cost of traversing readily to-be-garbage objects is reduced. Write-barrier is also better than read-barrier, since modifying data objects is less often than dereferencing pointers in most application programs.

In the context of 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 very 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 the 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 (arbitrarily variable size data unit) and buddy (system data unit of 2's power size).[2]

The main marking process (or main marker) traverses data objects from system's root such as global symbol package list and 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) marks data objects that have been notified by write-barrier. Both processes have its own marking stack. GC processes are also registered in the active process table, but their stacks will not be scanned, of course.

In the literature, contrary to our approach, sweeping is often neglected with respect to real-time GC algorithm. However, in order not to load symbolic processing primitives with GC overheads, it is important to separate the GC works from mutator primitives and load them to separate processes that can be very easily interleaved with mutators. This is the reason why sweeping is done by independent processes.

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 GC, there are three phases: *GC-off*, *mark*, and *sweep*. We call *mark* and *sweep* phases *GC-on* phase generically. In the GC-off phase, the GC processes are all sleeping. When one of the six data type categories is detected to be short, then GC wakes up. The shortage detection is done in each memory allocation primitive such as `cons` and `make-vector`. The remaining amount of free data objects is compared with *waterline* of the data type category, that is, a fraction of the total memory amount of that data type category, which can be adjusted dynamically by system parameters.

4.1 Mark phase

When 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 the 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 real-time GC are the active process table and runnable process queue. The former is a 64K 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. Any 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 cyclic list; therefore, enqueuing a process invokes `cons`.

After marking some other small system roots, the main marker scans the active process table sequentially from top to 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 starts. Here, a number of processes may have run during the first mark phase, whether it has been traversed or not. Under the principle of incremental update, it is enough to scan (maybe again) the stack of the process which has run after `activeT-front` has gone ahead of, or when it stays just at its entry address in the active

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

process table. We call such a process "clobbered" after the Lisp Machine Lisp Manual [Wein83]. Note that even in the second mark phase, new 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 process remains untraversed. Clobbered processes are sought in a 1K 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 yield a big inseparable section, there is one more 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 is zero. Bits in these bit-tables are set by the micro-kernel in the mark phase, and are cleared by the main marker. Note that the micro-kernel cooperates with GC very closely on these bit-tables.

To make clobbered process marking fast, 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, thereby its scan is omittable. 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-barrier in mark phase

In the mark phase, fields of already traversed and marked data object 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 GC, and the newly referenced object would be collected as garbage. The write-barrier is a well-known mechanism to prevent such accidental reclamation. As described above, stack write operation does not need a write-barrier.

In the TAO/SILENT microprogram, modification of a field of a data object is written, in principle, by a micro-subroutine call, classified as `wcad` routines. There are about thirty write-barrier subroutines that can fit a variety of register conditions and field conditions 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, whether writing into car or cdr matters. Writing into cdr is a little simpler than car, because it need not care about marking bit preservation.

```
(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) )
```

where `shallow-mark` is

```
(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) ))
```

Shallowly markable data include those which can be marked very 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 writer-barrier pushes 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 on 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 [Dijk76], posted data objects can be said colored gray, but it is seen as black from write-barrier, since the mark bit is set. To sum up, there are two marking processes and there is also a stack that stores gray objects.

4.3 Sweep phase

When both markers finish their work, GC enters the sweep phase, and the six sweeping processes (or sweepers) start running. 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.

The six sweepers correspond to the TAO data type categories: cell, vector, buddy, string body, symbol, and binary program blocks called *dytes*.

Sweeping proceeds from top to bottom for every data type category. Each sweeper has a variable representing its sweep frontier. It should be noted that free memory (or *chunks*) can exit ahead of the frontier, since 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-barrier in sweep phase

With our concurrent sweepers, write-barrier is needed also in sweep phase to prevent live data 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 prematurely.[3] The latter may not be, in general, called a write-barrier, but we include it here because the purpose is the same.

Writing into car, namely `wca` has to preserve the modified data's mark bit, because if it is set, the corresponding sweeper does not scan it yet. But writing into cdr need not do any check.

```
(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)
```

Allocation of a new data object ahead of the corresponding sweep frontier has to set its 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 the mark phase never sets the mark bit, so that very short lived cons cells created in the mark phase are more likely to be collected as garbage in the sweep phase.

4.5 Bypassing write-barrier

Some write operations in the system microcode do not need write-barrier. We have nine rules for omitting write-barrier check, thereby the write-barrier overhead are mitigated. For example, if it is known that the write operation writes only immediate data in cdr field (or equivalently, writes only immediate data into a vector — See 4.6), it is sure that no shallow-mark is needed in the 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 the micro-kernel process queue handling. Processes of the same priority are enqueued in a cyclic list of cell data type. When a

[3] Recall that our GC does not do copying or compaction.

process is to be dequeued from the queue, an operation such as `(setf (cdr x) (cddr x))` is needed.[4] 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 GC

Both markers can be ground up into very fine program segments as can be easily understood, considering parallel marker implementation with 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 state. 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 (nearly) 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.[5] 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 writes something in every reclaimed cell.[6]

Our cell sweeper, instead, collects contiguous free cells as one free chunk as large as the sweeper is not preempted by the scheduler. Only one write operation is performed per chunk (See Figure 1). For every live cell, however, the sweeper must erase 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 GC starts can easily merge with those newly reclaimed. From the allocator's view point, the free cell pointer into a free chunk can be cached in a register, and in many cases, `cons` has only to advance the free pointer and decrease the remaining size of the chunk 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.

---

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

[5] 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 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 real-time GC.

[6] Write operation takes more machine cycles than read operation in SILENT.

This quite straightforward improvement brought in some complexity in the logic of cell sweeper grinding, because the sweep frontier and `cons` frontier (i.e., 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 (with other complication of the memory management structure that is not mentioned in the paper).

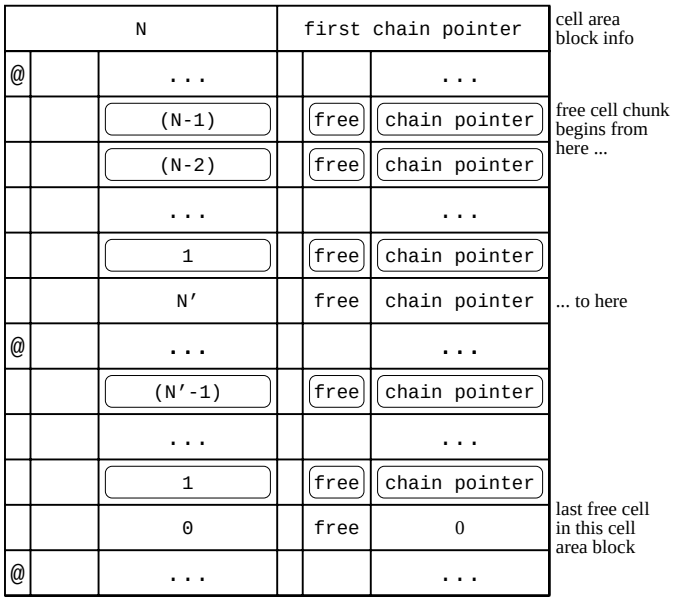


Figure 1. Internal structure of a free cell area

A cell area is 32K 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 copying and compactifying GCs.

4.7 No write as far as possible

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

Each vector has only one mark bit at its head's car, while its cdr contains the vector size. Writing 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 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 next vector (or chunk) head.

Each buddy has only two mark bits at its head'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 lighter write-barrier advantage. 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 car.

As the cell sweeper, to make sweeping 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 a "under initialization" status so that the marker does not go further into the vector or buddy elements. Thus, large data block initialization can be safely interleaved with other processes.

4.8 Hardware support and coding techniques

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

Write-barrier is implemented as a micro-subroutine which is called by the following microinstruction form:[7]

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

where `bsr` denotes multiple entry subroutine call to `wca`. One of `wca`'s three entries will be selected by the condition `gcm`. `gcm` is an alias of `psw3-2`, which means bit3 and bit2 of PSW (processor status word). SILENT PSW has four user definable bits (bit3-0). GC uses bit3-2 to represent one of the three phases of GC, and uses bit1 to distinguish the GC-off phase from GC-on phase. This implies that no explicit check action beforehand is needed to know which phase is at present. If `gcm` is used with a simple multi-way branch instructions `br` and `bsr`, the phase check is completely transparent (i.e., overhead-less) if no special action for 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 marker's microcode (including the shallow marker), 64-way branch case statements are used at seven places, 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 steps, thereby there is no room to check `hap`-interrupts (described soon later) in the loop. If there is, incredibly, no other data object reference in a 128 word D-block, 256 micro-steps (7.7 μsec) will run without interleave; this is one of the longest inseparable sections in GC.

There are a number of other small coding techniques around GC. For example, to grind up the cell sweeper thoroughly, the main sweeping loop 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 (0.12 μsec).

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 GC is invoked, and how the GC processes are scheduled among mutators.

The eight GC processes are scheduled slightly differently from mutators. Mutators are given the CPU time by a priority-based preemptive scheduling with round-robin; each process in the same priority queue is given an 8 $m$sec 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 (or *GC urgency*), instead. At present, the priority of all the GC processes is set to 4, an apparently lower priority which does not hinder important process scheduling even when it is referenced 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 interrupt facilities: 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 take

[7] As can be seen, microprogram itself is written in S-expression, which by nature enhances the flexibility and extensibility of the micro-assembler and linker.

place typically by the cause of 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 microcode explicit `hap` check. The causes of `hap`-interrupts are stack overflow, machine-cycle counter down-to-zero, specific 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 GC. Soon, a `hap` check detects the `simhap` and call the scheduler. This indirection for calling the scheduler is necessary since the allocation routine detects the memory shortage mostly deep in the 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 32K 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 routines.

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 GC to catch up with mutators.

5.3 GC urgency

In the GC-on phase, GC processes are scheduled according to parameters representing the urgency of GC. 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 8th scheduling, but for the highest urgency, GC processes run at every second scheduling; that is, GC processes and mutators run alternatively. Defaulted quota given to GC processes is 15.5 *m*sec at present. Thus, in most urgent situation, GC get twice as much CPU time as mutators. In the GC-on phase, GC urgency is dynamically updated and used for scheduling.

We think that much more experience is needed to fix the criteria of GC urgency and scheduling policy based upon the GC urgency.

6 Evaluation

We evaluated the GC performance in two ways: counting exactly on the program 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-cycle counting

Dynamic steps for write-barrier is the main overhead brought in to Lisp primitives in our implementation. The following table shows the number of machine-cycles for write-barrier that are not needed in stop mark-sweep GC.

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

where a range m–n means that m is minimum overhead and n is 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 object's slot and vector element has the same overhead as rewriting cdr as described in 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.

Process management of the micro-kernel suffers from only little overhead. For example, in the GC-on phase, additional 4 machine-cycles are needed to make a runnable mutator run. However, setting the bit-tables for a clobbered process takes 40 to 50 machine-cycles every time the clobbered process releases the CPU; this is the worst overhead.

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). It looks a little complicated, but it is a simple extension of recursive definition for Fibonacci function

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

Every recursive call is replaced by a process spawning and two returned values are received from the two 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. So a simple trick is embedded to waste cells and enlarge marking root.

(2) `cell-eater`

This is a benchmark to measure the GC speed rather than its response time (See Appendix B). `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 GC to catch up with the mutator since the mark phase takes more time and the mutator take 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.

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.

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

This benchmark holds big marking roots (processes and their stacks), but does not use so much cells. Here we set the size of cell area 533K cells to make GC-off phase and GC-on phase even. In this setting, one GC invocation has to scan as much as 1.6M words in process stacks, 95% of of which are swapped out when they are scanned. Stack dirty flags save 35% 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.

	time [μsec]	percentage	comments
elapsed time	1,149,427,948	100.00	
net computation	556,703,988	48.43	
micro-kernel	149,980,058	13.05	management of processes and stacks
GC processes	442,743,902	38.52	sum of all GC processes' CPU time
main marker	344,893,464	30.00	big root!
post marker	5,973,642	0.52	small for this benchmark
cell sweeper	78,826,828	6.86	
vector sweeper	2,038,127	0.18	
buddy sweeper	8,242,238	0.72	

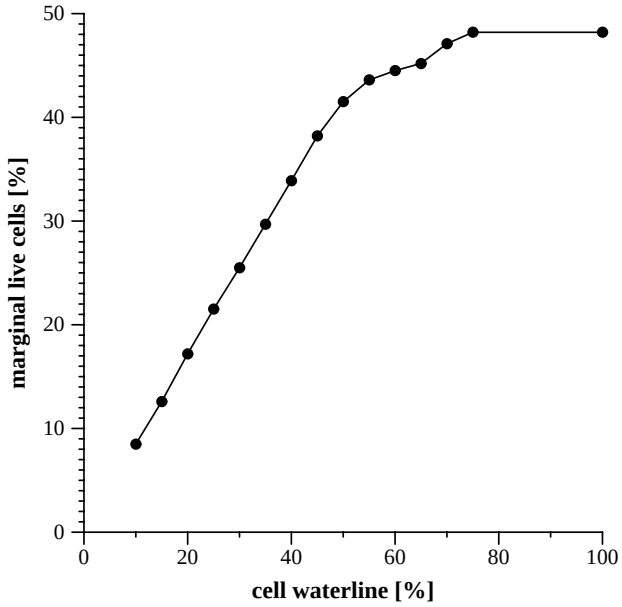
For bigger n, say 22, GC still catches up with those mutators, where the maximum number of concurrent processes is 34,817. We once ran this benchmark, by giving bigger second argument (loop count), for nearly a month without any trouble.

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

For *live* = 0 and 8M free cell area, we have the following result. 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.

	time [μsec]	percentage	comments
elapsed time	795,993,783	100.00	
net computation	604,767,619	75.98	
micro-kernel	1,859,869	0.23	nearly zero, of course
GC processes	189,366,296	23.79	about 1/4 of elapsed time
main marker	2,023,024	0.25	little
post marker	107	0.00	further little
cell sweeper	186,869,769	23.48	speed of cell reclamation
vector sweeper	30,655	0.00	
buddy sweeper	18,422	0.00	

The following 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 measurement, external events are raised every 33.3 *m*sec from the FEP (Front End Processor of SILENT) having animation graphics in mind.

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

Totally, 33,963 interrupts take place. As we said in 6.3, half of them take place in GC-off phase, and the other half in GC-on phase. The following table summarizes the resulted delay in μsec, where σ denotes standard deviation.

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

Contrary to most people's intuition (but so we anticipated), average wake-up delay is smaller in GC-on phase than in GC-off phase. 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 phase. An all day long measurement 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 worst 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 for 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 apart from this. In the following table (delay in μsec), we do not discriminate GC-off and GC-on phases, since they make little difference.

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

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.

7 Retrospection and Concluding Remarks

Early in the paper, 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 emergent work without consuming much amount of resources, its wake-up delay is the main concern, since even in GC-on phase, Lisp primitives such as `car` and `cons` do not suffer from GC overheads. Major wake-up delay is not derived from GC but from the micro-kernel and lengthy Lisp primitives. In other words, real-time processes will not be aware of GC at all; they can see micro-kernel and Lisp (self!) overheads only, however. This is exactly what we mean by the word "transparent."

The wake-up delay of this order (50–160 μsec) turned out to be 100 or 1,000 times smaller when we surveyed other related researches [Lim98, Huel98] in 1999. However, it should be noted that we planned this GC implementation in 1991 and announced that the target delay time is below 100 μsec [Take91]! The basic algorithm of this concurrent GC was implemented on TAO/ELIS, our former Lisp machine system, in mid 80's, but it was tested only partially so that we did not publish the result to the community. In fact, the name `wcad` is inherited from the TAO/ELIS microcode.

Quick survey revealed that wake-up delay or pause time has long been told in term of millisecond. We wonder why this simple idea, incremental update with write-barrier has not been realized so long time. Nothing very special is involved in our basic algorithm and implementation. We suspect this is a sort of "Columbus's egg"; that is, something simple or possible but which no one has ever tried actually or seriously.

However, the following can be said as the reason of our success:

(1) Inseparable sections can be easily ground up by virtue of two stage interrupt handling.
(2) We coordinate GC and operating system so intimately that GC can have its own scheduling policy and enjoy light-weight process mechanism.
(3) Our GC is simple because it is not parallel, but concurrent on a single processor; it is mark-sweep, not copying; and it does not employ compaction.
(4) A number of tiny programming techniques are piled up harmoniously.
(5) Some hardware features assist the implementation.

Some of these may be not so easy to port to other GC implementations on the current stock hardware. Considering the astonishingly growing micro-processor speed, however, we believe that the techniques of this simple-minded but somewhat tough-to-implement GC can be applied to future symbolic processing systems with much higher performance figures.

[REFERENCES]

[Bake78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communication of the ACM*, 21(4) pp.66-70, March, 1978.

[Boeh91] Hans-J. Boehm, Alan J. Demers, Scott Shenker. Mostly parallel garbage collection. *SIGPLAN Notices* 26(6), pp.157-164, ACM SIGPLAN, ACM Press, Toronto, Ontario, Canada, June 1991.

[Dijk76] Edsgar W. Dijkstra, Leslie Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in Cooperation. *Communications of the ACM*, 21(11), November 1978.

[Henr94] Roger Henriksson. Scheduling real time garbage collection, LU-CS-TR:94-129, Department of Computer Science, Lund University, 1994.

[Hibi90] Yasushi Hibino, Kazufumi Watanabe and Ikuo Takeuchi. 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, IPSJ, 1990.

[Huel98] Lorenz Huelsbergen, and Phil Winterbottom. Very concurrent mark-&-sweep garbage collection without fine-grain synchronization. *Proceedings of the First International Symposium on Memory Management*, pp.166–175, ACM Press, Vancouver, October 1998.

[Kung77] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. *Proceedings of the Eighteenth Annual Symposium on Foundations of Computer Science*, pp.120–131, IEEE, Providence, Rhode Island, USA, IEEE, New York, USA, October 1977.

[Lim98] Tian F. Lim, Przemyslaw Pardyak, and Brian N. Bershad. A memory-efficient real-time non-copying garbage-collector. *Proceedings of the First International Symposium on Memory Management*, pp.118–129, ACM Press, Vancouver, October 1998.

[Take86] Ikuo Takeuchi. Hiroshi G. Okuno, Nobusato Ohsato. A List Processing Language TAO with Multiple Programming Paradigms. *New Generation Computing*, Vol 4, No. 4, Ohmusha and Springer Verlag, 1986.

[Take91] I. Takeuchi, Y. Amagai, K. Yamazaki, and M. Yoshida. Real-time features of TAO/SILENT, IPSJ SIGSYM, 61-1, Sep. 1991 (in Japanese).

[Take00] I. Takeuchi, Y. Amagai, K. Yamazaki, and M. Yoshida. Lightweight processes in the real-time symbolic processing system TAO/SILENT, *Advanced Lisp Technology*, IPSJ, Gordon and Breach, 2000 (to appear).

[Wein83] Daniel Weinreb, David Moon and Richard M. Stallman. *Lisp Machine Manual*. Fifth Edition, System Version 92, LMI, 1983.

[Wils94] Paul R. Wilson. Uniprocessor garbage collection techniques, Technical Report, University of Texas, January 1994. Expanded version of the IWMM94 paper.

[Yuas90] Taiichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, pp.181–198, 1990.

Appendix A. Program `mfib-loop`

```
(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) make a process to execute a chore
          (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 root for marking bigger
; nevertheless, returned value it 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)))) )
```

Appendix B. Program `cell-eater`

```
(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 0))))))))
        (cons (list (list (list (list (list (list (list n)))))))
              (list-n (1- n))) ))

(defun sigma-list (l)
    (if (empty? l) 0 (+ (caaaar (caaaar l)) (sigma-list (cdr l)))) )
```