Return Barrier

Taiichi Yuasa,[†] Yuichiro Nakagawa,[†] Tsuneyasu Komiya[†]
and Masahiro Yasugi[†]

Garbage collection (GC) is the most popular method in list processing systems such as Lisp to reclaim discarded cells. GC periodically suspends the execution of the main list processing program. In order to avoid this problem, real-time GC has been proposed, which runs in parallel with the main program so that the time for each list processing primitive is bounded by some small constant.

The snapshot GC, which is one of the most popular real-time GC algorithms, has to mark all cells directly pointed to from the root area at the beginning of a GC cycle. The suspension of the main program by this root scan cannot be ignored when the root area is large.

This paper proposes "return barrier" in order to divide the process of root scan into small chunks and to reduce the suspension time of the main program. The root area on the stack is marked frame by frame each time a new cell is required. When a function returns, the garbage collector checks if cells pointed to from the frame of the caller function have been already marked, and marks them if not. After marking all cells directly pointed to from the root area, other cells are marked as in the original snapshot GC.

In this paper, we implemented the snapshot GC equipped with the return barrier in KCL (Kyoto Common Lisp). We compare and discuss the suspension times of this GC and the original snapshot GC.

1. Introduction

In most list processing systems, such as Lisp, data objects are represented by *cells*. A cell is allocated in the heap each time the application program (the *mutator*) requires a new data object. After a while, a cell may become *garbage*, i.e., it may no longer be used by the mutator. Since only a limitied amount of cells can be allocated in the heap, the system has to reclaim garbage cells and reuse them, when there remains no (or small) space available in the heap. The process to reclaim garbage cells is called *garbage collection* or GC for short. If a cell is pointed to directly or indirectly from the data area (the *root set*) that can be accessed any time to proceed computation, then the cell may be still in use. Therefore, garbage collection traverses pointers from the root set and reclaims only those cells that are not visited during the traversal. The reclaimed cells, which can be reused for further computation, are called *free cells*.

Most garbage collection is called *stop* garbage collection in that the system suspends computation of the mutator during garbage collection. Since the pause time is more than ten milliseconds in most cases, systems with stop garbage collection are not suitable for real-time applications. On the other hand, many realistic applications, such as those in intelligent agents and robot control, require real-time processing. In order to realize real-time list processing systems, real-time garbage collection is necessary, which does not suspend the computation for long.

On uni-processor machines, real-time garbage collection can be realized by splitting the entire GC process into small chunks and executing one chunk at a time along with computation of the mutator. Unlike stop garbage collection, real-time garbage collection has to cope with changes in the entire data structure of the heap, since the mutator keeps running even during garbage collection.

Algorithms so-far proposed for real-time garbage collection are characterized by their behavior against changes in the data structure. The snapshot algorithm[4),10),12)] developed by Yuasa requires special treatment, called *write barrier*, only when contents of cells are updated. Because of this feature, the algorithm has the following advantages over the other algorithms so-far proposed.

- Runtime overhead is kept relatively small, even without using dedicated hardware.
- It is relatively easy to implement in systems with conventional stop garbage collection.

† Graduate School of Informatics, Kyoto University

Thus the algorithm is suitable particularly for systems on off-the-shelf machines.

Snapshot garbage collection is based on the mark-sweep algorithm, which consists of two phases. The mark phase marks all cells that are pointed to directly or indirectly from the root set, and the sweep phase scans the entire heap to reclaim all unmarked cells as free cells. During the mark phase, the snapshot algorithm marks a small number of cells each time a new cell is requested by the mutator. During the sweep phase, it scans a small amount of the heap area each time a new cell is requested. In addition, at the beginning of each GC cycle, the snapshot algorithm takes a "snapshot" of the root set. That is, it scans the entire root set and marks all cells that are directly pointed to from the root set. This extra phase is called *root scan.* Even if a cell becomes garbage during a GC cycle, it is not reclaimed during that cycle. That cell will be reclaimed during the next cycle.

The weakness of the snapshot algorithm was that it suspends the mutator during root scan. Most of the suspension time is spent for scanning the stack, and the time for a stack scan depends on the stack size. Since the stack size varies dynamically, it is difficult to estimate how long a stack scan phase takes. In the worst case, the stack may be extremely long and the mutator's computation may be suspended for a long time.

In this paper, we propose a mechanism called *return barrier*, which is introduced to scan the stack incrementally. This mechanism is based on the following runtime mechanism, which is common to most modern programming languages.

- Program execution is performed by nested calls of functions (or methods).
- When a function is invoked, a new data area, called function frame, that is necessary to execute the call is pushed onto the stack. When a function returns, its frame is popped and the contents are discarded.
- While a function is being executed, it accesses only its own frame, but no other frames on the stack. (We will discuss exceptional cases in section 2.4.)

Return barrier has been implemented in several language systems, including Common Lisp[11)], Scheme[1)], multi-threaded Lisp[7)], and Java[8)]. In this paper, we report only the first implementation and show that return barrier is effective to reduce suspension times to the level that is acceptable for most real-time applications.

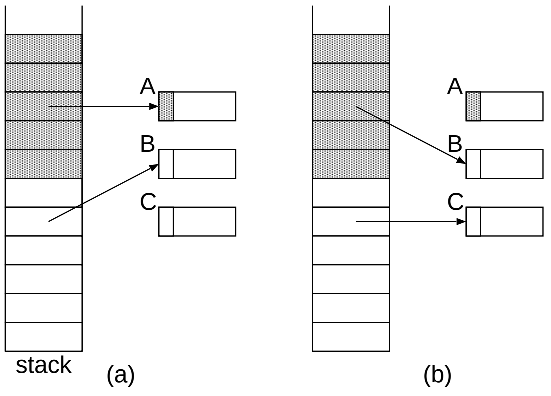

Fig. 1 Problem of incremental stack scan

2. Incremental Root Scan

The root set of a language system is divided into the stack area and the other area, called *static area.* The static area consists of system global variables such as the variable that contains a pointer to the symbol table. Since the static area is relatively small, it can be scanned at once without pausing the mutator for a long time. In Kyoto Common Lisp (KCL)[11)], for instance, the size of the static area that must be scanned consists of only 18 words, as we will discuss in section 4.1. This size of the static area is negligible when compared with the size of the stack area, which sometimes becomes one thousand words long. Therefore, we focus on how to scan the stack area incrementally.

2.1 The Problem

In order to obtain the same effects as stop stack scan, incremental stack scan has to guarantee that each stack entry be scanned before its content is modified by the mutator. Otherwise, some cells may be left unmarked after a mark phase, while they are still in use. Fig. 1 illustrates an example of such a situation. In the figure, gray stack entries are those already scanned and cells with gray "mark bits" are those already marked. Here we assume the stack is scanned downwards from the top to the bottom.

Consider the situation of Fig. 1 (a), where the stack has been partly scanned and cell A is already marked. Cell B is not yet marked because it is pointed to only from a non-scanned entry. Suppose that execution of the mutator is resumed in this status and the mutator changes the status as illustrated in Fig. 1 (b). Cell B

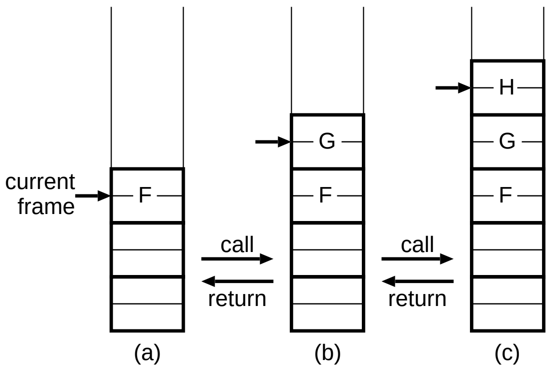

Fig. 2 Function frames

is still pointed to from the stack, but the entry that pointed to cell B in Fig. 1(a) now points to another cell C. After this, root scan will be continued, but cell B cannot be marked because it is pointed to only from an already scanned entry. As the result, cell B will be reclaimed even though it is still in use.

This situation could be avoided by using write barrier explained in section 1. However, write barrier for stack entries causes a large runtime overhead, mainly because every assignment operation to local variables becomes slow. Therefore, we would like to solve the problem without using write barrier. The key idea of our solution is to make use of the characteristics of function frames.

2.2 Function Frames

In general, program execution is performed by nested calls of functions. When a function is invoked, its function frame is pushed onto the stack. Each frame contains information that is necessary to execute the function call, such as arguments, local variables, and the return address. The frame of the function that is currently being executed is called the *current frame.* When a function returns, the current frame is popped from the stack. Fig. 2 illustrates this mechanism of function frames.

In Fig. 2 (a), function F is being executed and its frame is the current frame. When F invokes another function G, a new frame for G will be pushed onto the stack and it becomes the new current frame (Fig. 2 (b)). When G returns, its frame will be popped and the stack returns to the previous state before the call of G (Fig. 2 (a)). If G further invokes another function H, then a new frame for H will be pushed (Fig. 2 (c)), and the frame will be popped when H returns (Fig. 2 (b)).

For those programming languages that do not support local functions, execution of a function accesses only the current frame at the top of the stack. The other frames below the current frame are not accessed until control returns to the corresponding function and the frame becomes the current frame. This implies that the problem described in the previous section can be solved without write barrier to the stack, if it is guaranteed that the current frame always resides in the part of the stack that have been scanned already.

2.3 Return Barrier

Return barrier is a mechanism to guarantee that the current frame resides in the scanned part of the stack at any time during root scan. It is placed at the "front line" of root scan, i.e., at the bottom of the scanned part of the stack. When a function returns, if a function frame below the barrier is to become the new current frame, then the return instruction will be trapped. Control then transfers to a special routine, called *barrier code.* The barrier code proceeds stack scan for some amount of stack entries including those in the new current frame, and then moves down the barrier to the new front line. After this process, the trapped return instruction is executed and the computation of the mutator is resumed.

Fig. 3 shows how return barrier works. In Fig. 3 (a.1), the current frame of G resides in the scanned part of the stack. Since execution of G affects only the current frame, the system would not fail to mark used cells. However, when G returns to the caller F, the frame of F, which is below the front line of stack scan, becomes the new current frame (Fig. 3 (a.2)). If control returns to F immediately, then execution of F may cause a problem. In order to avoid this situation, the return barrier is set at the front line. The role of the barrier is to trap the return instruction from G and to transfer control to the barrier code. After the barrier is moved down (Fig. 3 (b.2)), control returns to the caller F, this time the new current frame of F being within the scanned part of the stack (Fig. 3 (b.3)).

Return barrier can be implemented without any runtime overhead, if the system can access the return address stored in function frames. When return barrier is placed at the bottom of a function frame of F, the system saves the return address into a fixed location and overwrites the return address in the frame with the address of the barrier code. Then, when F

returns, control automatically transfers to the barrier code. At the end of the barrier code, the system retrieves the saved return address and transfers control to that address. With this implementation of return barrier, it is not necessary for functions to check whether the barrier code should be invoked on their returns. This means their compiled code need not be changed to cope with return barrier.

Note that at most one return barrier may be placed in a stack. Therefore, single-threaded systems require only one location to save the overwritten return address. For systems that support multiple threads, each thread has its own stack and thus one location is necessary for each thread. The best place would be in the thread structure, which contains all information necessary for multi-threading.

With the mechanism of return barrier, incremental stack scan is performed as follows. At the start of a GC cycle,

- scan the entire static root area that need to be scanned, and
- scan the current frame and place return barrier at the bottom of the current frame.

During the root scan phase, each time a new cell is requested,

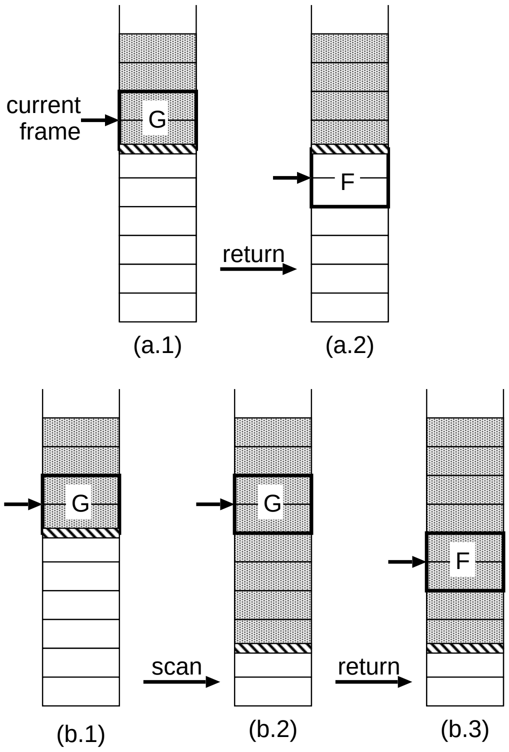

Fig. 3 Return barrier

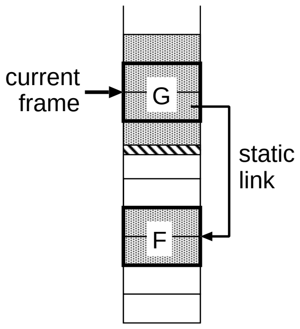

Fig. 4 Processing local function frames

- scan a certain number of frames, and
- if the last frame at the bottom of the stack has been scanned, shift to the mark phase,
- otherwise, place return barrier at the bottom of the last scanned frame.

Incremental scan in multi-threaded systems can be done in a similar way. Major differences are:

- scan all the current frames of active threads at the start of a GC cycle and place return barrier at the bottom of each current frame, and
- shift to the mark phase only when all stacks have been scanned.

2.4 Remarks for Implementation

So far, we have assumed:

- execution of a function accesses only the current frame, and
- functions always return to the caller.

In real systems, these conditions are not always satisfied. Systems that support local functions do not satisfy the first condition and those that support non-local exit do not satisfy the second condition. In this section, we discuss these exceptional cases and show return barrier can be used efficiently for those systems.

Consider the following function definition in Common Lisp[9].

```
(defun F (x)
  (labels ((G (y z) ··· (G z x) ···))
    (G x x)))
```

In function `F`, local function `G` is defined, and `G` invokes `G` itself recursively. Variable `x` in the expression `(G z x)` is the parameter to `F` and is allocated in the frame of `F`. Thus, during execution of `G`, `G` accesses not only `G`'s frame but also `F`'s frame. In order to make this access possible, `G`'s frame contains the static link to `F`'s frame (see Fig. 4).

In general, during execution of a local function, some frames accessible through the static link may be located below return barrier as in

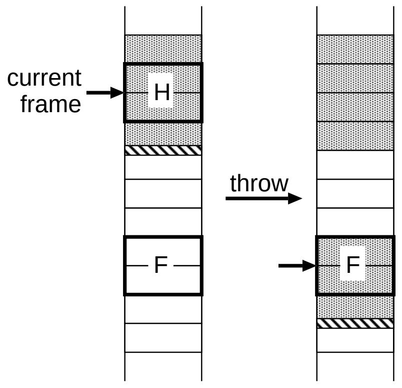

Fig. 5 Processing non-local exit

the case of Fig. 4. Therefore, when the system scans the frame of a local function, it should scan all frames that are accessible from the frame through the static link.

This special handling of local function frames does not cause an obstacle to real-time processing, since the number of frames that are accessible from a local function through the static link does not exceed the nested level of that local function.

Non-local exit transfers control to a certain function that satisfies some condition, rather than to the caller. Catch & throw is a typical non-local exit mechanism. Fig. 5 illustrates the behavior. Consider the case where function F establishes a catcher and then a throw is executed that corresponds to the catcher during execution of function H (Fig. 5 (a)). By throwing, the normal mechanism of function call and return is skipped, and control returns directly to function F from function H (Fig. 5 (b)).

If a non-local exit unwinds the stack beyond return barrier, the new current frame (F's frame in the example) must have been scanned before control transfers to the catcher function. In general, the destination frame (i.e., the new current frame) of a non-local exit is dynamically searched. In the case of catch & throw, the system searches the stack for a catcher with the same tag given to the throw, from the top towards the bottom. During this search, the system can determine whether the stack is unwound beyond return barrier. If so, the only extra work is to scan the new current frame. The point here is that, we do not need to scan those frames between the old current frame and the new current frame. The number of such frames is unbounded, and if the system scans them all, it may fail real-time processing. However, since these frames are never accessed, no problem will arise even though they are skipped without being scanned. Skipping some frames means the algorithm is not strictly a snapshot any more. This is because those cells that are reachable only from skipped frames are reclaimed during the current GC cycle. Nevertheless, skipping frames is desirable, since it only causes some garbage cells to be reclaimed earlier.

3. When to Initiate

In real-time garbage collection, the mutator keeps requiring new cells even during garbage collection and free cells are consumed. However, garbage cells are reclaimed only during the sweep phase. If there remain too few free cells at the start of a GC cycle, the system may run out of free cells before it starts reclaiming garbage cells. This situation is called *starvation.* If the system falls into starvation during garbage collection, it has either to expand the heap or to complete the rest of the GC cycle at once. These "emergency processes" may pause the mutator for a long time. Therefore, the system should initiate a GC cycle while there remain enough number of free cells. In this section, we discuss a sufficient condition to avoid starvation. We use the following system parameters.

K_0: the number of entries on the stack to be scanned at each cell request during the root scan phase

K_1: the number of cells to be marked at each cell request during the mark phase

K_2: the number of cells to be swept at each cell request during the sweep phase

Fig. 6 gives a rough graph of $F_q(t)$, the number of free cells of type q when the mutator requests the t-th new cell after the start up of the system. In the figure, we assume a GC cycle starts at time a, when the number of free cells of type q reaches a certain number M_q. Then at time b, the mark phase terminates and the sweep phase starts, and the GC cycle ends at time c.

When garbage collection is initiated, the system first scans root incrementally. No garbage cells are reclaimed during this root scan phase. As new cells are requested, the number of free cells decreases. Let $R(a)$ be the stack size at time a. Then the number of cell requests during root scan is:

$R(a)/K_0$

Root scan is finished at time $t = a + R(a)/K_0$, and the mark phase starts. Since no garbage

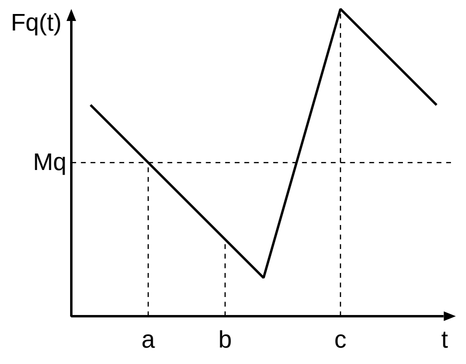

Fig. 6 Number of free cells

cells are reclaimed during the mark phase, the number of free cells keeps decreasing. Let $A(a)$ be the number of used cells at the start of the GC cycle. Since K_1 used cells are marked at each cell request, the number of cell requests during the mark phase is:

$A(a)/K_1$

Therefore, we obtain the following equation.

$b = a + R(a)/K_0 + A(a)/K_1$

The mark phase is finished at time b and the sweep phase starts. Let N_q be the total number of type q cells in the heap and $A_q(a)$ be the number of used cells of type q at time a. Then the number of type q garbage cells that are reclaimed during the sweep phase is:

$N_q - A_q(a) - F_q(a)$

Usually, these garbage cells are scattered over the entire heap, but in an extreme case, they all resides at the end of the heap. Therefore, at most

$(N - (N_q - A_q(a) - F_q(a)))/K_2$

cell requests will be made before the system begins to reclaim garbage cells of type q. If cell requests of type q cells happen at the frequency rate of C_q, then before the system starts reclaiming type q garbage cells, at most

$$C_q\ (R(a)/K_0 + A(a)/K_1 \\ +(N - N_q + A_q(a) + F_q(a))/K_2)$$

free cells of type q will be consumed after the start of the GC cycle. If there remain more than this number of type q free cells, then the system does not fall into starvation. Thus we obtain the following sufficient condition to avoid starvation.

$$F_q(a) \geq C_q\ (R(a)/K_0 + A(a)/K_1 \\ +(N - N_q + A_q(a) + F_q(a))/K_2) \tag{1}$$

In order to determine when to start a GC cycle in real systems, let us give estimated values for C_q, $R(a)$, $A(a)$, and $A_q(a)$.

First, if we assume that $C_qA(a)$ is equal to the number of used cells $A_q(a)$ of type q.

$$C_qA(a) = A_q(a) \tag{2}$$

We also assume a clever memory allocation so that more frequently requested cell types have more cells in the heap than less frequently requested cell types. That is, the number of type q cells in the heap is proportional to the frequency rate C_q of type q cell requests.

$C_q = N_q/N$

Then from the equation (2), we obtain:

$A(a) = NA_q(a)/N_q$

$A_q(a)$ is hard to estimate because it depends on when the system initiates a GC cycle. Nevertheless, we need an estimation of $A_q(a)$ to determine when to start a GC cycle. We use the value of the worst case, where all type q cells other than free cells are in use.

$A_q(a) = N_q - F_q(a)$

It is impossible to estimate the stack size $R(a)$ at time a, since it depends not only on when to start a GC cycle, but also on application programs. Again, we consider the worst case, where all used cells are pointed to from the stack.

$R(a) = A(a) = N(N_q - F_q(a))/N_q$

With these estimated values, the condition (1) is transformed to:

$$F_q(a) \geq \frac{N_q(1/K_0 + 1/K_1 + 1/K_2)}{1 + 1/K_0 + 1/K_1}$$

In other words, the system can avoid starvation when the number of type q free cells becomes:

$$M_q = \frac{N_q(1/K_0 + 1/K_1 + 1/K_2)}{1 + 1/K_0 + 1/K_1}$$

For instance, in case $K_0 = K_1 = K_2 = 20$,

$M_q = 0.1364N_q$

That is, the system should start a GC cycle only when the number of free cells of type q becomes less than 14% of the q cells in the heap. On the other hand, with the original snapshot algorithm that scans the entire root at once, $K_0 = \infty$ and we obtain:

$M_q = 0.0952N_q$

Snapshot garbage collection with return barrier should be initiated earlier than the original snapshot because of the difference of these two values of M_q. Note that since we assumed $R(a) = A(a)$ in the above calculation, garbage collection is initiated much earlier than necessary. Nevertheless, we observe the difference with the original snapshot will be acceptable.

4. Implementation on KCL

In this section, we report our first implementation of return barrier in KCL (Kyoto Common Lisp)[11]. We first describe the root set of KCL and then we discuss the problems we encountered and our solutions to them.

4.1 The Root Set of KCL

The root set of KCL consists of the static area and the stack. The static area is actually a vector of 160 cell pointers. Most elements of this vector is initialized during the start-up of KCL and remain unchanged. These elements are used to store data objects, such as the `nil` object, that are used directly by the system and to protect them from being reclaimed by garbage collection. Only 18 elements of the vector change their values at runtime. These elements are used as system-internal temporary variables. Only these 18 elements need to be scanned at the start of a GC cycle. The other constant elements can be scanned little by little, since no write access is made to them.

KCL uses the following five stacks.

- the value stack, in which local variables and temporary variables for compiled functions are allocated, and through which parameters and return values are passed.
- the bind stack, which is used to implement shallow binding of dynamic variables. Each entry is a pair of the name of a dynamic variable and its previous value.
- the frame stack, which stores catchers for non-local exit that may be initiated by `throw` or `return-from`. Each catcher contains necessary information to resume computation after the non-local exit, including stack pointers of the other stacks when a catcher is established by `catch` or `block`.
- invocation history stack, which contains the history of function calls and is use for debugging.
- the C language stack. The compiler of KCL translates Lisp code into C language code and generates object code by using a C compiler. The kernel of KCL is also written in the C language. Therefore, execution of KCL is controlled by using the C language stack. In order to provide a high portability, KCL never accesses the C stack directly.

Among these stacks, the C stack is not subject to root scan. Although cell pointers may be stored in the C stack, such pointers are always stored also in the value stack. In this way, garbage collection is made independently from the structure of the C stack.

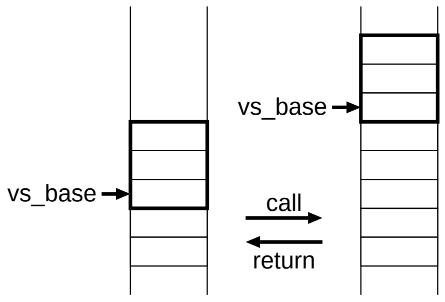

Fig. 7 The value stack

4.2 Implementation of Return Barrier

Since return addresses in KCl are stored in the C stack, they cannot be overwritten with the barrier code address. Therefore, we cannot apply the efficient method described in section 2.3 to implement return barrier. Instead, we adopted the following algorithm for each stack other than the C stack.

- When garbage collection is initiated, scan the 18 roots that may change their values dynamically, and scan the current frame of each stack.
- Each time a cell is requested, scan each stack for a certain number of entries.
- When a function returns, check whether the new current frames of each stack has been scanned. If not, scan the stack until the entire new current frame has been scanned.
- When a non-local exit is initiated by `throw` or `return-from`, scan the new current frame of each stack.

With this algorithm, whenever a function returns, the system has to check whether the new current frame has already been scanned. This causes an overhead on function execution, rather than root scan itself. Therefore, the system performance is reduced compared with the algorithm in section 2.3.

In order to realize this algorithm, a variable named `barrier` is prepared for each stack. Each `barrier` variable points to the front line of stack scan of the associated stack. Return barrier is implemented for each stack as follows.

The current frame of the value stack is the stack area between the base pointer `vs_base` and the stack top (see Fig. 7). When a function returns, `vs_base` moves down towards the bottom of the stack. When this occurs, the system checks whether the new value of `vs_base` becomes below `barrier`. If so, execution of the mutator is suspended and the system does the following process.

- If there are K_0 or more entries between `vs_base` and `barrier`, then scan them all and moves `barrier` to the position of `vs_base`.
- If there are less than K_0 entries between `vs_base` and `barrier`, then scan the stack for K_0 entries and moves down `barrier` for K_0 entries.

In the former case, the number of entries to be scanned depends on application programs. There is the possibility that the mutator be suspended for long. However, that number does not exceed the number of local variables and parameters that can be accessed at the same time and thus it can be regarded to be relatively small in most applications.

The bind stack does not need return barrier. When the bind stack is popped, the previous value of the dynamic variable is stored in the value slot of the symbol object that names the dynamic variable. Thus write barrier guarantees the popped value will be eventually marked.

Also, the invocation history stack does not need return barrier. This stack contains pointers to active functions, but these pointers are never overwritten nor stored in the root area.

Each catcher in the frame stack contains the lexical environment which is used by the interpreter to execute non-compiled functions. When a non-local exit is thrown, the interpreter retrieves this lexical environment to resume computation. Since lexical environments of KCL are represented by cons cells, they are subject to marking. Thus, before the frame stack is unwound, the system checks whether the catcher resides below `barrier` and if so, scans the catcher.

5. Performance Evaluation

In order to evaluate the performance of return barrier, we ran several benchmark tests on the version of KCL with return barrier and compared the results with those on the version of KCL with original snapshot GC without return barrier. In this section, we report the resluts of the following two typical benchmarks tests.

fib

calculates the n-th Fibonacci number by recursive calls. When this program is run on the KCL interpreter, two cons cells are consumed each time a function is invoked, to build a lexical environment. These cells become garbage after the function returns. We use the results for $n = 27$ for performance evaluation. For $n > 27$, KCL automatically expands the heap and the result is not reliable for performance evaluation. For $n < 27$, the results are similar to the case of $n = 27$.

boyer

proves logical formulas[2]. This program consumes a large number of cells and accesses them frequently. It requires a large size of the stack, changes the stack size heavily, and invokes garbage collection many times.

The machine we used for the tests is a Solaris machine with 200 MHz Pentium Pro processor and 192 MBytes of physical memory. In order to avoid automatic heap expansion during test execution, we allocated 1.5 MBytes of heap area and forced garbage collection before each test run, so that all garbage cells have been reclaimed. Time is measured by using the RDTSC (read time-stamp counter) instruction as recommended by Intel Corporation[5].

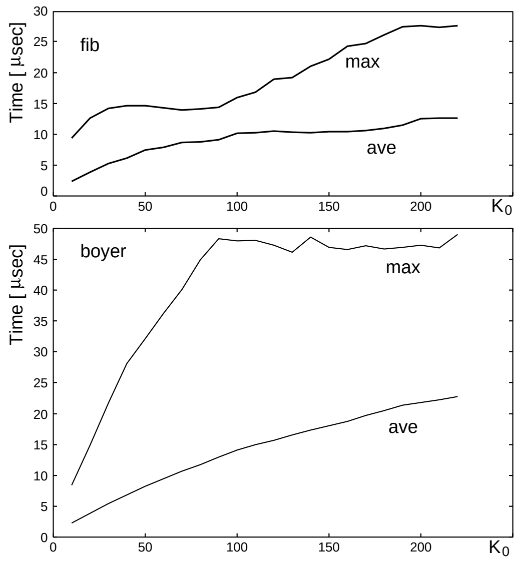

Fig. 8 Suspension times during root scan

5.1 Suspension Times

Return barrier is intended to reduce suspension time of the mutator by scanning the root set incrementally. If we use a smaller number for K_0 (the number of stack entries that are scanned at each cell request), the suspension time should become smaller. To see the effect of the value of K_0, we measured the suspension times for various values of K_0. Fig. 8

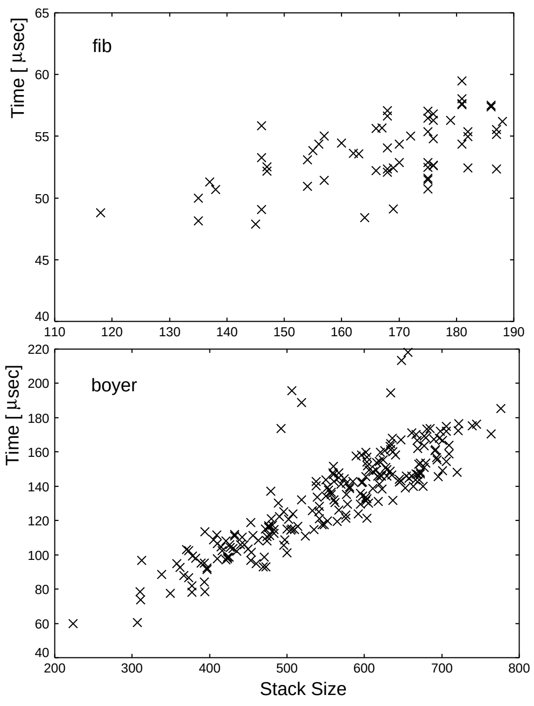

Fig. 9 Stack sizes and root scan times

shows the maximum suspension times and average suspension times for $K_0 = 10, 20, \ldots, 220$.

From the figure, it is clear that average suspension time decreases in proportion with the value of K_0. Maximum suspension time shows the same characteristics, but the relation with K_0 is not so strong as average suspension time. Also, for each K_0, it is observed that the difference between maximum suspension time and average suspension time is not small. This difference is regarded as the result of cache effects, since the same number of entries are scanned each time.

For comparison, we measured times for root scan on the version of KCL without return barrier, i.e., suspension times caused by scanning the entire root at once. Fig. 9 shows the relation between the stack size and the time for root scan. Each point corresponds to one garbage collection cycle. Here, the stack size is the total number of entries of the four stacks of KCL that are subject to root scan. From the figure, it is observed that the stack size at the start of a GC cycle changes heavily. In the boyer benchmark, the largest stack size is four times larger than the smallest.

The relation between the stack size and the scan time may seem not very strong. This is perhaps because of the cache effects. However, we can see the scan time is approximately proportional to the stack size. The suspension times for root scan is as follows.

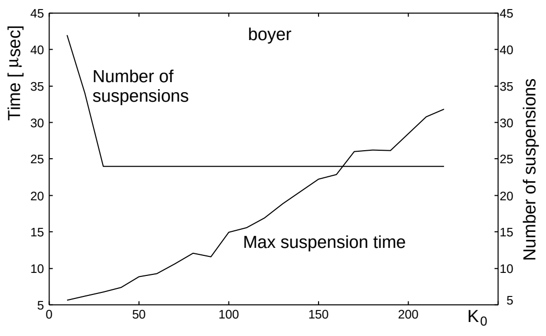

Fig. 10 Suspension times caused by return barrier

	maximum	average
fib	58.3 μsec	53.0 μsec
boyer	218.5 μsec	145.3 μsec

With return barrier, suspension time depends on the parameter K_0, but if $K_0 = 10$, which is the smallest value we used, maximum suspension time is less than 10 μsec and average suspension time is less than 5 μsec for both benchmarks. Thus we can conclude return barrier reduces suspension time to a great extent.

5.2 Suspension by Barrier Trap

When the stack unwinds beyond return barrier, the mutator process is suspended and the system scans the roots in the new current frame. We measured the number of such suspensions and the maximum suspension time. The result is that for the fib benchmark, no such suspension occurred. The result for the boyer benchmark is given in Fig. 10.

With return barrier, each time a cell is consumed, return barrier moves down toward the bottom of the stack. If several functions return consecutively without requesting new cells, return barrier may suspend the mutator. Even in such a case, however, if root scan has already been finished or if return barrier has moved near the bottom of the stack, then no suspension may occur. Thus the possibility of suspension depends on programs and on runtime status. It seems a coincidence that no suspension occurred during execution of fib. In such a case, return barrier on the stack is never used and should be regarded as a kind of "safety insurance", since it is impossible to predict whether return barrier is used or not.

On the other hand, in the case of boyer, return barrier caused suspensions. In case $K_0 = 10$, the mutator was suspended 42 times. As we will see later in Table 2, execution of boyer caused 250 GC cycles. So, mutator suspension occurs once for six GC cycles and we can say the possibility is very low. Also, it is clear that the suspension time is quite short. Obviously, the maximum suspension time decreases as the value of K_0 decreases. On the other hand, the number of suspensions tends to be larger for smaller values of K_0. This is because the speed of return barrier to move downwards is proportional to K_0.

5.3 Overhead

In order to evaluate the runtime overhead of return barrier, we measured the execution time of benchmark programs, the number of GC cycles, and the average time for root scan. The results are shown in Tables 1 and 2. In these tables, we list the results for $K_0 = 10, 20, 40, 80, 160$ for garbage collection with return barrier. For snapshot GC without return barrier, we list the results on the "check added" system as well as on the system with the original snapshot. We will later explain what "check added" means.

In all these case, the system marks used cells and sweeps the heap incrementally. Thus the benchmark results depends on the parameters K_1 (number of cells to be marked at a time during the mark phase) and K_2 (number of cells in the heap to be swept at a time during the sweep phase), as well as on K_0. Since the purpose of this experiment is to see the overhead of root scan, we used the fixed values $K_1 = K_2 = 40$.

Runtime overhead of return barrier is caused by the following reasons.

(1) Garbage collection cycles should be initiated earlier than the snapshot algorithm. This causes more GC cycles.

(2) Control frequently transfers between the mutator and the root scan routine because the entire process of root scan is interleaved with the mutator process. This causes more cache mishits.

(3) When a function returns, if the new current frame resides below return barrier, then the mutator will be suspended and the system proceeds root scan for a certain amount. This causes another control transfer overhead.

(4) In our implementation on KCL, we could not apply the technique to overwrite return addresses. Each time a function returns, it must make a barrier check explicitly.

As we discussed in the previous section, (3) does not cause a large overhead, since the possibility of barrier traps is very low. (4) is specific to our implementation, and therefore, we wanted to obtain benchmark results without this overhead. For this purpose, we prepared another version of KCL by adding an explicit barrier check each time a function returns. The results on this version are listed in the "check added" columns in Tables 1 and 2. This version resulted in the same number of GC cycles and the same time for root scan as the original snapshot, but longer execution time because of the extra checks. The difference of the execution times is considered as the overhead of (4). The difference is only 2% for boyer, but 10% for fib. This is because each call of the fib function makes only a small amount of job and thus the time for extra checks is relatively large.

Since the number of GC cycles of fib is relatively small, even though GC cycles are initiated at different timings, it is not reflected in the number of cycles. The difference of execution time is caused by the reason (2) above. Compared with "check added", the difference is 3% for $K_0 = 10$, but only 0.5% for $K_0 = 160$. The time for root scan differs depending on K_0, but the total time for root scan (average time multiplied by the number of GC cycles) is only 5 msec for $K_0 = 10$ and will not affect the total execution time. Therefore, the difference of the total execution time is mainly caused by control transfer.

For boyer, a large number of cells are consumed and garbage collection is initiated frequently. Thus the timing of GC initiation strongly affects the number of GC cycles and the difference is reflected to the total execution time. For $K_0 = 10$, the number of GC cycles is 50% more than for $K_0 = 160$, and the total execution time is 33% more. In order to apply return barrier to real systems, K_0 should be chosen carefully by considering the tradeoff of smaller K_0, which reduces suspension time, against larger K_0, which reduces total execution time. On the other hand, for $K_0 = 160$, return barrier and snapshot do not make a large difference both on the number of GC cycles and on the total execution time. In this experiment, the suspension time is 50 μsec (see Fig. 8) for $K_0 = 160$, which is short enough for many real-time applications.

Table 1 Execution results of fib

	with return barrier					without return barrier	
	$K_0 = 10$	$K_0 = 20$	$K_0 = 40$	$K_0 = 80$	$K_0 = 160$	check added	original
total execution time [sec]	12.270	12.110	12.030	11.980	11.980	11.920	10.830
number of GC cycles	65	65	65	65	65	65	65
ave. root scan time [μsec]	81.127	72.745	67.063	64.483	63.908	52.975	52.975

Table 2 Execution results of boyer

	with return barrier					without return barrier	
	$K_0 = 10$	$K_0 = 20$	$K_0 = 40$	$K_0 = 80$	$K_0 = 160$	check added	original
total execution time [sec]	36.380	32.470	28.710	27.790	27.380	26.680	26.110
number of GC cycles	304	268	226	216	212	208	208
ave. root scan time [μsec]	192.556	163.640	156.794	150.465	146.436	143.744	143.744

6. Related Research

Kondo proposed an incremental root scan algorithm[6]. His algorithm also scans the stack for a certain number, say K, of entries at a time, but the direction of the scan is from bottom up. Bottom-up scan is adopted in the hope that the stack may become shorter while the lower part of the stack is scanned. After each scan of K entries, the next K entries are protected against write access from the mutator. If the mutator tries to write into the protected area, then it is suspended and the protected area is scanned. This algorithm has the following problems.

- Bottom-up scan is difficult to implement for many systems, in which function frames in the stack can be located only by traversing frame links downwards. In such systems, stack scan requires an additional means to distinguish ordinary data objects from return addresses and frame pointers in the stack.
- The mutator suffers from a large overhead, since it has to check whether the destination is write-protected each time it writes into the stack.

Iwai improved the algorithm[3] by using another direction for stack scan. In addition, the improved algorithm makes use of the paging mechanism of the underlying operating system in order to avoid the overhead on the mutator. This algorithm in turn has the following problems.

- It depends on the operating system, and thus cannot guarantee real-time processing.
- The system cannot be portable, since it contains code that depends on the paging mechanism of the operating system.

7. Conclusions

We proposed an algorithm to scan the root set incrementally, by introducing return barrier to snapshot garbage collection. We have implemented the proposed algorithm onto KCL, and showed that the algorithm actually reduces suspension time of the mutator to a desirable level. Because of the reasons described in section 4.2, we cannot apply the efficient implementation technique in section 2.3 for KCL. Instead, we simulated the effect of the technique and showed the overhead of return barrier can be kept quite small.

The proposed algorithm scans the stack for a small amount each time a cell is requested. If functions return consecutively, without requesting new cells, several function frames will be popped from the stack and return barrier may suspend the mutator. For some extreme programs, the mutator is suspended periodically, and in the worst case, the system fails real-time processing of the application. However, the result of our experiments shows that the possibility of the mutator's suspension by return barrier is quite small. We believe we have established a real-time algorithm that is effective for realistic applications.

References

1) M. Flatt: PLZ MzScheme: Language Manual, `http://download.plt-scheme.org/doc/mzscheme/`.
2) R. Gabriel: *Performance Evaluation of Lisp Systems*, MIT Press (1985).
3) T. Iwai and M. Nakanishi: Reduction of Pause Time due to Snapshot Parallel GC (in Japanese), *Journal of Information Processing*, Vol. 40, No. SIG4 (May. 1999).
4) R. Jones and R. Lins: *Garbage Collection*, John Wiley & Sons (1996).
5) Intel Corporation: Using the RDTSC Instruction for Performance Monitoring, `http:`

`//www.intel.co.jp/drg/pentiumII/appnotes/RDTSCPM1.HTM`.

6) G. Kondo and M. Nakanishi: Efficiency Improvement of Root Insertion on Real-time Garbage Collection (in Japanese), IPSJ SIG Notes, PL No. 16 (Nov. 1997).
7) T. Matsui and M. Inaba: Euslisp: An Object-Based Implementation of Lisp, *Journal of Information Processing*, Vol. 13, No. 3 (1990).
8) Omron Information Technolory Research Center: JeRTy, `http://www.jerty.com/jerty_e/index.html`.
9) G. Steele: *Common Lisp the Language, Second Edition*, Digital Press (1990).
10) T. Yuasa: Real-time garbage collection on general-purpose machines, *The Journal of Systems and Software*, Vol. 11, No. 3, pp. 181–198 (Mar. 1990).
11) T. Yuasa: Design and Implementation of Kyoto Common Lisp, *Journal of Information Processing*, Vol. 13, No. 3 (1990).
12) T. Yuasa: Real-time Garbage Collection (in Japanese), *Information Processing*, Vol. 35, No. 11, pp. 1006–1013 (Nov. 1994).
13) T. Yuasa, Y. Nakagawa, T. Komiya, and M. Yasugi: Return Barrier, *Journal of Information Processing*, Vol. 41, No. 9 (2000).