Mark DURING Sweep rather than Mark THEN Sweep

PARLE '89, LNCS 365, Springer-Verlag, June 1989.

Christian Queinnec*
LIX
École Polytechnique
91128 Palaiseau cedex
France
queinnec@poly.polytechnique.fr

Barbara Beaudoing
LIX et SAGEM
beaudoin@poly.polytechnique.fr

Jean-Pierre Queille
SAGEM
Chaussée Jules César
95520 Osny
France

Abstract

Garbage Collection frees the programmer from the burden of explicitly deallocating unused data. This facility induces a considerable overhead but also causes some delays that may affect real-time applications. Guaranteed throughput (with at most short and predictable delays) is needed in many applications such as plane or plant control and requires at least a worst case analysis to identify the performances of the whole system. Traditional GC are made of two phases : the marker which identifies all useful data, followed by the sweeper which reclaims all useless data. On-the-fly GC schemes were introduced to permit an application and a collector to run concurrently. That concurrency may lessen the GC penalty incurred by the application.

We present here a new algorithm where the application, the marker and the sweeper are concurrent. The benefit is to tightly adjust collection rate to application consumption and have an allocation time bounded by a small constant. Moreover our algorithm does not waste memory and appears to be well suited for embedded systems.

This "mark DURING sweep" algorithm is completely presented. An interesting single-processor and incremental realisation is also analysed and followed by some implementation variations.

1 Introduction

The technique known as *Garbage Collection* frees the programmer from the burden of explicitly deallocating unused data. Traditional languages such as Lisp [McCarthy 60] or modern ones such as Smalltalk [Goldberg 83] or ML [Milner 84] use a Garbage Collector (GC). It had been noted that a substantial part of execution time – ten to thirty percent – is spent in this activity. Improvements in GC are worth doing since they mainly contribute to overall system quality. Unfortunately, efficient GC are the most difficult part to achieve in such languages.

Many algorithms exist which can be sorted into various families [Cohen 81]. We follow here the taxonomy used in Lang and Dupont [Lang 87]. *Reference-counting* uses the number of references to a datum; when this number falls to zero, the datum is useless and its storage can be reclaimed. The *tracing* family identifies all useful data as being accessible from a root data set; the unreachable data are garbage and can be reclaimed.

*This work has been partially funded by Greco de Programmation.

Tracing algorithms can be subdivided into *mark-and-sweep* or *copy* collections. Moreover *stop-and-collect* [Fenichel & Yochelson 69] or *incremental* [Baker 78] versions exist for both.

Multiprocessor architectures introduced distributed collections : many processors scavenge concurrently separate data spaces [Hudak & Keller 82, Hughes 85, Derbyshire 88]. *On-the-fly* collections [Dijkstra 78] are parallel collections : the collector runs concurrently with the evaluator. The former reclaims unused data created and then lost (i.e., made unreachable) by the other.

Performance of these algorithms depends on many parameters such as :

- the ratio of the number of used cells to the total number of cells in the memory [Appel 87]
- the need to compactify the memory to prevent thrashing or to linearize accesses [Bobrow 79]
- the presence of a virtual memory system [Moon 84]
- the detection of particular policies of cell consumption such as stack allocation [Queinnec 88], pools of resources [Symbolics Reference Manual], weak pointers [Rees 84]
- etc.

The goal of this paper is to present a new parallel collection algorithm. It was designed for real-time applications in embedded systems. Embedded systems usually have narrow memories since these memories are expensive, take space and power, induce weight and heat. CPU throughput is (or should be) designed for worst case, it is usually (at least at the beginning of the life cycle) large enough to guarantee fast response time. Hence our proposed algorithm had to fulfill the following requirements :

short and predictable response time :
time lag between events and corresponding handlers must be very short ($\approx 50\mu s$),

guaranteed throughput : to avoid unexpected GCs,

narrow memory : input/output hardware and memory still form the main part of computers prices.

Stop-and-copy collections waste memory. Asynchronous page defaults occur within virtual memory system and thus induce unpredictable delays incompatible with these above real-time requirements[1].

In traditional mark-and-sweep and in stop-and-collect mode, sweeping can be lazy and deferred onto each allocation performed by the evaluator. After the marking phase is complete, the evaluator is immediately resumed. When it has to allocate a cell, the evaluator sweeps as much of the memory as needed to find an unused cell. Note then that a free-list is not necessary. This technique smoothes the response time but the longest delay is due to the marking phase.

On-the-fly collection [Dijkstra 78] introduces two concurrent processes known as the mutator and the collector. At least two realisations are possible. In the first one, each process is run on a separate processor. Thus collection is done by a dedicated processor, but some specific hardware (tags or FIFO or common registers etc.) is needed by the two processors to cooperate. The other realisation makes use of a single processor. Since the two processes are concurrent, they can be freely intermixed. That gives an incremental version : a small amount of the collector process is done each time an allocation or a waiting-loop is performed by the mutator. This technique was already used for copy collections notably in Baker's version [Baker 78]. The main interest is for real-time uses. Whenever a high priority event occurs, the collector can be immediately stopped to the benefit of the mutator which can run the handler associated to this event. The response time is directly linked to the length of the longest critical section of the collector. Nevertheless mutator starvation is possible if the free-list is empty while the collector is still in marking phase. The mutator has then to wait for the end of this phase in order to get its allocation honoured.

Our proposed algorithm is a real mark *and* sweep named "mark <u>DURING</u> sweep". Marking and sweeping are performed concurrently and therefore leads to a fine-grained continuous mode where the consumption of the mutator is balanced by the collector (in a non-stop mode) without any fits and starts. This guarantees that, given a task which performs allocations, an upper time bound can be computed for its execution. This knowledge permits the analysis of overall response time for real-time applications and improves their safety.

[1] If some of the above requirements were to be relaxed, [Appel & Ellis & Li 88] would provide the most up-to-date alternative.

2 Mark DURING sweep

Let us define more precisely some terminology before explaining our algorithm. The memory, or the heap, consists of cells containing immediate data or pointers referencing cells. The roots are the data always needed by the evaluator that must persist after collections. The registers, the evaluation stack(s) and the free-list belong to the roots. The free-list is the list of cells that can be given back to the application by allocation requests. The collector has to identify unused cells and to append them to the free-list. Unused cells are the cells proved to be unreachable from the roots and reachable cells are either the roots or cells pointed to by reachable cells.

The evaluator executes the application. It uses cells, follows pointers, modifies them to designate other cells and allocates new ones. Given that the free-list belongs to the roots, the only modification performed by the evaluator is to redirect an *edge* i.e., a pointer from a reachable cell to another reachable cell. To reduce the evaluator to this sole job justifies its name given in [Dijkstra 78] : *mutator*. Note that the mutator is only interested in reachable cells while the collector is concerned by all cells.

The collector is composed of a marker and a sweeper. The marker has to identify all reachable cells. These cells may be marked by a bit in the header of the cell or a bit in a bitmap, among other techniques. We retain the colours used by Dijkstra : *a cell is black if marked and white if not.* The sweeper considers all cells in the memory, checking wether they are reachable or not and collecting the unused ones.

To make the mutator and the marker concurrent requires some cooperation between them. Whenever the mutator redirects an edge it has to signal the new target of the edge to the marker. This can be performed by a pushdown-list [Steele 75] or by a *grey bit* as in [Dijkstra 78]. The meaning of the grey bit is "the mutator guarantees this cell to be reachable : it must be marked and so must be its sons". The grey bit was introduced to permit the mutator and the collector to respect the following invariant :

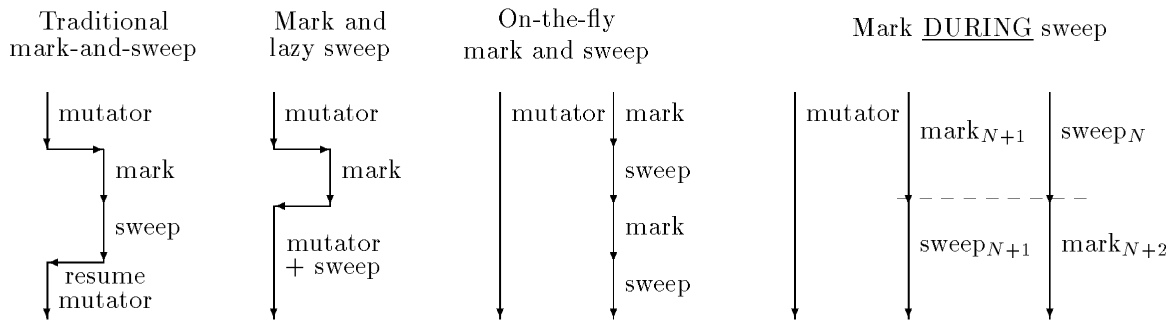
No black cell can point to a white cell. (P1)

Grey cells serve as intermediaries between black and white cells. They indicate where marking is still in progress. When all grey cells vanish, marking phase is over. Other palettes exist with two or four colours [Ben-Ari 84, Kung & Song 77].

These explanations show that the sweeper makes use of the marks left by the marker at the end of its work : black cells are reachable, white cells are garbage. That sequentiality seems to preclude concurrent marking and sweeping. Our solution is to consider two generations[2]. When the marking of the first generation is finished, sweeping that generation is started while marking the second is concurrently launched. This scheme works since sweeping modifies only unreachable cells of the first generation (which cannot be reachable at the same time in the second generation) and thus cannot interfere with the works of the mutator or the marker during the second generation. After sweeping the first generation and marking the second one is done, the collector can initiate a new cycle i.e., sweeping the second generation and marking the third. Hence two generations at most are simultaneously dealt with by the collector. Of course the physical marks (black, grey or white) used for the two simultaneously considered generations must not be confused. Nevertheless a single free-list is maintained independently of any generations.

Invariant (P1) is still valid if restricted to the generation of the marker.

We sum up this presentation by the following figure expressing the diversity of mark-and-sweep collections.



[2] The word "generation" is not to be confused with the use made by [Lieberman 83] which qualifies cells having comparable lifetimes.

3 The Algorithm

The algorithm is composed of three parts : the mutator, the marker and the sweeper, the last two form the collector.

Conventions

In order to simplify the algorithms, we consider cells to be all the same and to contain exactly two sons called left and right. Our algorithms can be easily extended to cope with varisized cells. Cells may be defined as :

```
const M { the total number of cells } ;
type cell : record
  left, right :  0..M-1 ;
  colour_N   : (white_N, black_N) ;
  colour_N+1 : (white_N+1, grey_N+1, black_N+1)
  endrecord
```

In our algorithm, variables **k** and **i** will represent indexes to cells in the heap i.e., numbers from **0** to **M-1**, where **M** is the total number of cells. The variable **c** will be a colour (black, grey or white) and **G**, a generation number. Generations will be denoted by indices: N or $N+1$. Atomic sections will be emphasized by hooks as in <atomic>.

The following sub-functions will be used.

left(i), **right(i)** returns, respectively, the left or right son of the $\mathtt{i}^{th}$ cell. If used in the left-hand side of an assignment, **left** and **right** respectively alters the left or right son of the $\mathtt{i}^{th}$ cell.

$\mathtt{colour}_N$**(i)** returns or alters the colour of the $\mathtt{i}^{th}$ cell of the heap according to generation N. The colour$_N$and colour$_{N+1}$functions are completely disjoined. Modifying the colour$_N$of a cell does not affect the colour$_{N+1}$ of this cell, and reciprocally.

$\mathtt{shade}_N$**(i)** turns a white cell to grey if not, otherwise does nothing. This modification is performed under generation N. Its precise definition is :

```
Shade_N(i) is { always atomic }
  begin c :=  colour_N(i) ;
        if c = white_N then colour_N(i) := grey_N endif
        end
```

Our algorithms will also contain commentaries or assertions expressed as { **assertion** }. The main invariant (P1) is : given a generation N, no black$_N$cell can directly point on a white$_N$cell.

The Marker

The marker algorithm is classical. All actions are indexed by a fix generation number, let us say here $N+1$. The marker looks for a grey cell in the heap, shades its sons, then blackens this very cell. It stops when no grey cell can be found.

```
Mark_N+1 is
  begin { no black_N+1 cell }
    forall i in roots do
       <shade_N+1(i)> endforall ;
    k := 0 ;
    while  k < M do
       <c := colour_N+1(k)> ;
       case c
         grey_N+1  : <shade_N+1(left(k))> ;
                     <shade_N+1(right(k))> ;
                     <colour_N+1(k) := black_N+1> ;
```

```
                k := 0 ;;                          { rescan heap }
        white_{N+1},
        black_{N+1} : k := k+1 ;;
      endcase
    endwhile
  end { no grey_{N+1} cell and all white_{N+1} are garbage_{N+1} }
```

Note that termination of the marker is ensured since grey cells become black, black cells remain black and black cells number is bounded by M. Moreover invariant (P1) is respected throughout the marker code. Unreachable cells present in the heap at the beginning of the marker will be left white at its end.

More efficient markers can be designed, see section 5, but this simple one simplifies proof.

The Sweeper

The sweeper has to collect all unused cells and append them to the free-list. As suggested by its name, it linearly sweeps the heap from one end to the other and examines the mark of all cells. The following sweeper is described in the context of the N^{th}generation.

```
Sweep_N is
  begin { no grey_N cell and all white_N cells are garbage_N }
    k := 0 ;
    for k from 0 upto M-1 do
      { ∀i < k ( garbage_N is reclaimed and colour_N(i) is white_N ) }
      <c := colour_N(k)> ;
      case c
        white_N : <append cell k to free-list ;shade_{N+1}(k)¿ ;;
        endcase
      <colour_N(k) := white_N> ;
      endfor
    end { all cells are white_N and garbage_N is reclaimed }
```

The main difference between a standard sweeper and this one is that appending an unused cell to the free-list is followed by a shading. This shading is performed under next generation i.e., the generation of the (con-)current marker. This action is very important since to forget it would violate invariant (P1) of generation $N+1$. In effect : the free-list belongs to the roots and must be marked (shaded) accordingly. Unused cells are white$_N$and since they are unused at generation N, they are also unused at generation $N+1$ and will be left white$_{N+1}$by mark$_{N+1}$. If these unused cells were appended to the free-list, one might create a black$_{N+1}$on white$_{N+1}$edge from some cell of the free-list (probably the last cell) to the newly appended cell.

Note that, after sweeping, all cells are white$_N$since used cells are whitened$_N$in order to satisfy the assumptions of the marker$_{N+2}$. Note also that the grey$_N$colour is completely irrelevant. grey$_N$cells may only appear as the result of a shade$_N$function. In the context of sweep$_N$, no call to shade$_N$is possible since the other tasks are the marker$_{N+1}$and the mutator which can only perform shade$_{N+1}$calls (cf. collector hereafter).

Due to its definition, the termination of the sweeper is obvious. Moreover white$_N$cells present in the heap at the beginning of sweeper$_N$are guaranteed to be reclaimed.

The Collector

The collector is responsible for launching and synchronizing markers and sweepers. The marker$_{N+1}$ and the sweeper$_N$must be launched simultaneously. The next collector cycle (i.e., marker$_{N+2}$ and sweeper$_{N+1}$) must wait the termination of both of them before getting control.

The construct `parbegin ... parend` [Brinch Hansen 73] means that all contained instructions are executed concurrently. Control is returned after all instructions are finished.

```
Collector is
  begin { all cells are white_N and white_{N+1} }
    <G := 0> ;
```

```
mark_0 ;
forever do
   parbegin mark_{G+1} ;
            sweep_G parend ;
   <G := succ(G)>
   endforever
end
```

Since the marker and the sweeper terminate, the `parbegin ... parend` construct must also terminate. For every N the collector cycles over a couple of marker$_{N+1}$and sweeper$_N$according to the previous figure.

We intentionally explained the marker in the context of the $N+1^{\text{th}}$generation while the sweeper was given in terms of the N^{th}generation. It appears that interactions are restricted to the management of the free-list which is shared between the mutator and the sweeper. This is the only mutation performed by the sweeper that requires the cooperation of the marker since these newly appended cells must be shaded$_{N+1}$as explained before.

The various colour states of the cells and the associated transitions will be thoroughly addressed later on.

The Mutator

The mutator must respect essentially the same invariant as in Dijkstra's algorithm : whenever an edge is redirected, the target of this edge must be shaded. The difference here is that shading is performed under $G+1$, the current generation of the concurrent marker.

```
Mutator is
    ...
    <left-or-right(k) := i ; shade_{G+1}(i)>
    ...
```

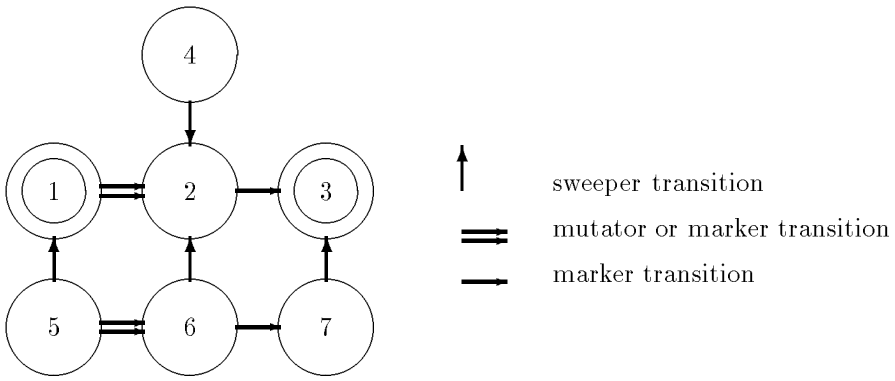
Case Analysis

Since there are at most two concurrent generations used by the marker$_{N+1}$and the sweeper$_N$, only six pairs of colours are possible for cells, (remember that grey$_N$cannot be encountered). The meanings of these states are summed up in the following table (state numbers refer to the next figure) :

	white$_N$	*black*$_N$
white$_{N+1}$	if swept$_N$ then (1) used$_N$ else (4) unused$_N$	(5) used$_N$
grey$_{N+1}$	(2) used$_{N+1}$	(6) used$_N$and used$_{N+1}$
black$_{N+1}$	(3) used$_{N+1}$	(7) used$_N$(but still not swept$_N$) and used$_{N+1}$

Used$_N$(resp. used$_{N+1}$) means that the cell is reachable during generation$_N$(resp. generation$_{N+1}$). Except (white$_N$,white$_{N+1}$) cells which can be in two states depending on the progress of the sweeper, all other pairs of colours uniquely identify a well defined state.

Transitions between these states are the following :



Remark that, for the sweeper, terminal states (when all cells are swept$_N$ and whitened$_N$) are (1), (2) and (3). Terminal states for the marker (without grey$_{N+1}$ cell) are (1), (5), (3) and (7). Therefore terminal states for each cycle of the collector are : (1) corresponding to used$_N$ but unused$_{N+1}$ cells and (3) corresponding to used$_N$ and used$_{N+1}$ cells. State (4) is a temporary state for cells which are garbage$_N$ but still not yet swept$_N$; cells in state (4) will end up to state (2) when they will be reclaimed.

It appears that only three bits are needed to record these states as shown in the record definition of `cell` given above. A clever technique can achieve good use of these three bits. Notice that two bits are required for $N+1^{\text{th}}$ generation while only one is needed for N^{th} generation. Since we have only two concurrent generations, when the collector starts a new cycle, the $N+1^{\text{th}}$ generation becomes the last one. The new one, numbered $N+2$, just begins its marking phase and needs two bits. The $N+1^{\text{th}}$ one now requires only one bit. So we can arrange the three bits in the following manner. One bit for grey$_{N+1}$, one bit for white$_N$ or black$_N$, one bit for white$_{N+1}$ or black$_{N+1}$. At each new cycle of the collector, the “grey” bit deserves the new generation and is only used by the current marker. Since at the end of the marker, no grey cell can exist, all the grey bits are in a steady state and can be reused. Thus the following encoding can be used :

Transitions		bit_N	bit_{N+1}	$grey_{G+1}\,bit$
	$black_N\,cell$	means black	–	–
5→1, 6→2, 7→3	$black_N \rightarrow white_N$	becomes white	is untouched	is untouched
	$white_{N+1}\,cell$	–	means white	is off
1→2, 5→6	$white_{N+1} \rightarrow grey_{N+1}$	is untouched	is untouched	becomes on
2→3, 6→7	$grey_{N+1} \rightarrow black_{N+1}$	is untouched	becomes black	becomes off

One can easily see from this table that the grey bit is only transient information only used during the marking phase. The grey bit can be reused without reinitialisation at each collector cycle.

4 An Incremental Garbage Collector

Since the three tasks (mutator, marker and sweeper) are concurrent, an incremental variant of these algorithms may be elaborated on a single-processor application. During each wait-loop or at each allocation performed by the mutator, a time-slice can be devoted to mark$_{N+1}$ a little and sweep$_N$ a little. The length of these time-slices can be parameterized according to the instantaneous allocation rate of the mutator. This incremental version, interlacing the three tasks, looks like Baker’s incremental GC where copying is spread over the mutator. Moreover since the commutation is done synchroneously in well defined parts of the mutator, the overhead for mutual exclusion is therefore lessened.

The overall parameters governing this process are

the total memory size (`M`), thus there are 2∗`M` pointers,

the consumption speed of the mutator (v_E) expressed in number of consumed cells by unit of time,

the reclamation rate of the sweeper (r_S) expressed by the ratio of reclaimed cells to swept cells by unit of time,

the instantaneous length of the free-list (L) which acts as a buffer between the sweeper and the mutator and smoothes local variations of r_S or v_E,

the rate of the marker (r_M) expressed by the ratio of marked cells to visited cells by unit of time.

The rate of the sweeper (r_S) and the consumption speed of the mutator (v_E) may be considered as grossly constant. Conversely the rate r_M of the marker decreases as the marking phase progresses. The overall constraint is that the marker must not finish after the sweeper which must itself finish before the mutator exhausts the free-list. More formally stated :

The mutator starvation is prohibited while garbage cells still exist.

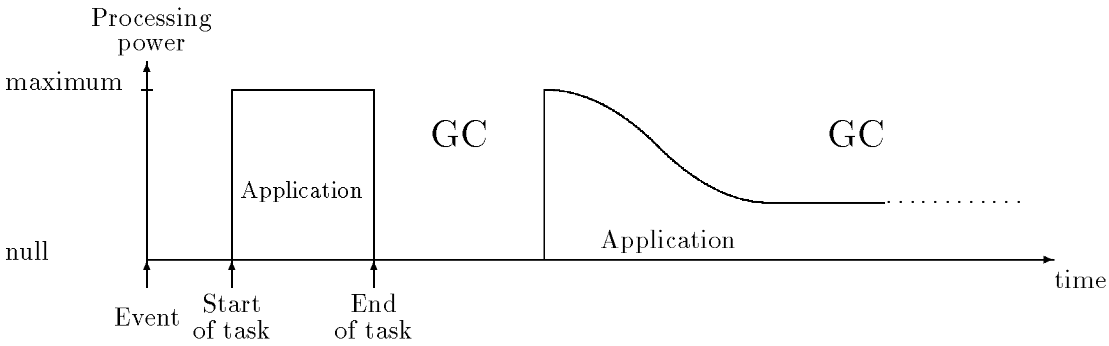
Garbage cells are unused white$_N$ cells not yet swept$_N$ or (and it is a stronger requirement) garbage$_{N+1}$ cells not yet identified by marker$_{N+1}$. The correctness criteria can nevertheless be ensured. The work of the marker cannot exceed the examination of 2*`M` pointers. Similarly the sweeper has only and exactly `M` cells to consider. We may adjust to `M`$/L$ and for each time-slice, the number of cells that the marker and the sweeper must visit. Note that now the marker cannot finish after the sweeper since they have a same speed : it will probably finish well before if garbage cells are present since these cells do not need to be marked.

When the mutator wants to allocate the last cell of the free-list, the sweeper will have then to examine `M` cells and so will the marker. `M` cells is the whole memory but not a whole cycle : the problem is to deal with cycles boundaries. Whenever the memory is completely swept$_N$, the current cycle will be ended by asking the marker$_{N+1}$ to finish its work. A new cycle is then started and the sweeper$_{N+1}$ and the marker$_{N+2}$ are asked to visit again `M`$/L$ cells where L is the new value of L.

Time devoted to GC is regulated by the length of the free-list, with the extra (but at most double) work at end of cycles. The speed of the mutator is thus proportional to the size of the free-list and inversely proportional to the size of the memory it occupies.

Two kinds of real-time tasks may be identified. The first one consumes a fixed amount of cells less than `M`. Given that the whole memory is in a steady state, such a task can then be launched without GC and uses all CPU resources. Collection will be done afterwards. Cyclic tasks such as sensor sampling belong to this kind of task. Other tasks may not be bound in terms of consumption. They must be run with the collector but a minimal processing speed can be guaranteed corresponding to an exact balance between the collector and the mutator.

The following figure summarizes these behaviours.



Tasks may be associated to a "profile" which the collector can use to tune the collection while the task is active. In the previous figure the left task does not require GC while the right one needs it. The part of the processing power devoted to this second task evolves to a steady state where consumption and production can be equated.

5 Variants

Several implementation variants of our algorithm were investigated. Some results follow.

- Since there are only two simultaneous generations in the whole system, we can use a flip-flop rather than incrementing `G`. We only need to modify function `succ` as :

```
type flip-flop is 0..1 ;
var G is flip-flop ;
function succ(G : flip-flop) returns flip-flop is
  begin if G = 0 then return (1) else return (0) endif ;
```

Similarly to Baker's, the function `succ` performs what can be called a *flip*. The flip has to reassign the grey bit to the new generation and to permute $\text{bit}_N$and bit_{N+1}meanings.

- Instead of shading_{N+1} the cells appended to the free-list, the sweeper can directly blacken_{N+1} them. This has two benefits. First, the free-list does not need to be marked (after the first generation) since cells are already $\text{black}_{N+1}$when appended. Second, since it does not introduce any more gray cells it induces faster terminations of the marking phase. The proof of our algorithm is not modified.

Tricks like identifying $\text{black}_N$and white_{N+2}colours [Hudak & Keller 82] i.e., exchanging the meaning of these colours between two generations acting on the same physical bits may be further investigated.

- The atomic section performed by the mutator can be refined into :

```
Mutator is
    ...
    <left-or-right(k) := i> ;
    <shade_{G+1}(i)>
    ...
```

As shown by Van de Snepscheut, the order of operations is important [Van de Snepscheut 87]. As also shown in [Dijkstra 78], the proof must be revised since we now violate invariant (P1). Between the two operations, there exist an edge from a $\text{black}_{G+1}$to a possibly white_{G+1}cell. Invariant (P1) must be refined to support this temporary and local configuration.

- Remark that in this marker definition, shading each target of a cell (its left and right sons) is atomic. Targets of large objects such as vectors of pointers or hash arrays, can be progressively shaded. Therefore we avoid the penalty of atomically copying that large object in a non-stop copy collection [Baker 78]. Nevertheless the problem still arises if we need to compactify the heap.

- Scanning the memory again and again is a pain. Each time the marker finds a grey cell, it has to rescan the whole heap to check if there are any other grey cells. A global counter can be maintained to record the total number of grey cells in the heap. The sub-function $\texttt{shade}_N$ is now

```
shade_N(i) is { always atomic }
  begin c :=  colour_N(i) ;
        if c = white_N then colour_N(i) := grey_N ;
                            total-grey-counter := total-grey-counter + 1
                            endif
        end
```

and the "`while k < M`" loop of the marker may now be written as "`while total-grey-counter > 0`". Note that when this counter falls to zero, it cannot change at all; if every reachable cell is marked, the job of the marker is finished and the mutator cannot introduce new grey cells (remember that the free-list is marked_N : free cells are reachable !). The only liberty (and duty) we have is to reach this steady state with a non-empty free-list to avoid mutator starvation.

- Another improvement is to mark recursively as much as possible the useful data and then "hunt" the few remaining grey cells. The benefit is to lessen the number of cells to be examined while marking. The counterpart is to use a stack (which may be bounded [Knuth 68]) to record cells which right offspring is to be marked. The marker then becomes

```
MarkN+1 is
  local function scan(k) is { assume colourN+1(k) = greyN+1 }
         begin <shadeN+1(left(k))> ;
               scan(left(k)) ;
               <shadeN+1(right(k))> ;
               scan(right(k)) ;
               <colourN+1(k) := blackN+1>
               end ;
  begin { no blackN+1 cell }
        forall i in roots do
           <shadeN+1(i)> ; scan(i) endforall ;
        k := 0 ;
        while total-grey-counter > 0 do
           k := find-a-grey-cell(k) ;
           scan(k)
           endwhile
        end { no greyN+1 cell and all whiteN+1 are garbageN+1 }
```

- Bitmaps can also be used to record where grey cells are. The grey bits of cells are packed together to form a large string of bits called the *grey bitmap*. Similarly $\text{bit}_N$and bit_{N+1}bitmaps can be formed. These bitmaps ease the marker and the sweeper. Instead of rescanning the whole memory and look for grey cells, the marker can use the grey bitmap to more precisely locate where are grey cells (cf. function `find-a-grey-cell` above). Furthermore the bit_Nbitmap can be scanned by the sweeper to check reachability. The main interest is that bitmaps can be efficiently analysed word after word with appropriate instructions on stock hardware. Bitmaps of bitmaps can also be done to ease bitmap handling. These solutions may also be hardwired.

6 Conclusions

We presented a new algorithm that real-time embedded systems can use. It was designed to fulfill the following requirements : short and predictable response time, guaranteed throughput and narrow memory.

Our algorithm called "mark DURING sweep" proceeds by running a marker and a sweeper concurrently to the mutator. Sequentiality of the sweeper with respect to the marker is achieved by working on two different generations : after one generation is marked, it is given to the corresponding sweeper while marking a new generation is initiated.

Interlacing the three processes provides an incremental version where the mutator consumption may be balanced by the sweeper production given that the marker proceeds concurrently to prepare the next instantaneous flip. This system is therefore reduced to a control problem. That version of "mark DURING sweep" fits the needs of embedded real-time systems. An implementation on a 1024-processor machine is under progress at SAGEM. The machine is intended to satisfy the needs of intelligent control process on planes or robots.

References

[Appel 87] Andrew W. Appel, *Garbage Collection can be Faster than Stack Allocation*, Information Processing Letters 25 (1987) 275–279.

[Appel & Ellis & Li 88] Andrew W. Appel, John R. Ellis, Kai Li, *Real-time Concurrent Collection on Stock Multiprocessors*, Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 22-24, 1988, pp 11–20.

[Baker 78] H.G. Baker, *List Processing in Real Time on a Serial Computer*, Communications of ACM, Vol. 21, No 4, April 1978, pp 280–294.

[Ben-Ari 84] Mordechai Ben-Ari, *Algorithms for On-the-fly Garbage Collection*, ACM Transactions on Programming Languages and Systems, Volume 6, Number 3, July 1984, pp 333 – 344.

[Bobrow 79] Daniel G. Bobrow, Douglas W. Clark, *Compact Encodings of List Structure*, ACM transactions on Programming Languages and Systems, Vol. 1, No. 2, October 1979, pp 266–286.

[Brinch Hansen 73] Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[Cohen 81] Jacques Cohen, *Garbage Collection of Linked Data Structures*, Computing Surveys, Volume 13, Number 3, September 1981, pp 341 – 367.

[Derbyshire 88] Margaret H. Derbyshire, *Mark Scan Garbage Collection On A Distributed Architecture* submitted to Lisp And Symbolic Computation.

[Dijkstra 78] Edsger 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, Volume 21, Number 11, Novembre 1978, pp 966 – 975.

[Fenichel & Yochelson 69] R.R. Fenichel, J.C. Yochelson, *A Lisp Garbage Collector for Virtual-Memory Computer Systems*, Communications of ACM, Vol. 12, No 11, November 1969.

[Goldberg 83] Adele Goldberg and David Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1983.

[Hudak & Keller 82] P. Hudak and R. Keller, *Garbage Collection and Task Deletion in Distributed Applicative Processing Systems*, ACM Symposium on Lisp and Functional Programming, August 1982.

[Hughes 85] John Hughes, *A Distributed Garbage Collection*, Lecture Notes on Computer Science, Vol. 201, Springer-Verlag, Functional and Programming Languages and Computer Architecture, Nancy, France, September 1985.

[Knuth 68] Donald E. Knuth, *The Art of Computer Programming, Fundamental Algorithms*, Addison-Wesley, 1968.

[Kung & Song 77] H.T. Kung and S.W. Song, *An Efficient Garbage Collection System and its Correctness Proof*, Proceedings of IEEE 18th Symposium on Foundations of Computer Science, October 1977, pp 120–131.

[Lang 87] Bernard Lang, Francis Dupont, *Incremental Incrementally Compacting Garbage Collection*, SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques, Saint Paul, MA, pp 253 – 263.

[Lieberman 83] Henry Lieberman and Carl Hewitt, *A Real-time Garbage Collector based on the Lifetimes of Objects*, Communications of the ACM, 23(6), 1983, pp 419–429.

[McCarthy 60] John McCarthy, *Recursive Functions of Symbolic Expressions and their Computation by Machine – Part I*, Communications of ACM, Vol 3, N 1, 1960, pp 184–195.

[Milner 84] Robin Milner, *A Proposal for Standard ML*, ACM Symposium on Program and Functional Programming, 1984, pp 184–197.

[Moon 84] David A. Moon, *Garbage Collection in a Large program System*, 1984 ACM Symposium on Program and Functional Programming, Austin, Texas, pp 235–246.

[Queinnec 88] Christian Queinnec, *Dynamic Extent Objects*, Lisp Pointers, Vol. 2, No 1, 1988.

[Rees 84] Jonathan A. Rees, Norman I. Adams, James R. Meehan, *The T Manual*, Fourth Edition, 10 January 1984, Computer Science Department, Yale University, New Haven CT.

[Van de Snepscheut 87] Jan L.A. Van de Snepscheut, *"Algorithms for On-the-fly Garbage Collection" revisited*, Information Processing Letters 24, 1987, pp 211 – 216.

[Steele 75] Guy L. Steele Jr., *Multiprocessing Compactifying Garbage Collection*, Communications of the ACM, Volume 18, Number 9, September 1975, pp 495 – 508.

[Symbolics Reference Manual] *Symbolics Reference Manual*, Symbolics.