Incremental Multi-threaded Garbage Collection on Virtually Shared Memory Architectures

Thierry Le Sergent, Bernard Berthomieu

Laboratoire d'Automatique et d'Analyse des Systèmes du CNRS
7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
e-mail: lesergen@laas.fr, bernard@laas.fr

Abstract: This paper describes a multi-threaded and incremental garbage collector operating on shared memory architectures. The technique was developed for parallel implementations of the language LCS, a high level parallel programming language.

An incremental, page trap based, collection algorithm operates locally on each of the processors. Processors alternatively plays the role of mutator and collector. The processors cooperate for collection and mutation; idling processors perform part of the collection task for the others until they acquire some work. The progress of collectors versus allocations is controlled by a scan credit mechanism that guarantees a responsive execution of the application. There is no static partitioning of the storage among the processors; pages are dynamically allocated to any processor, for specific purposes.

Two implementations are discussed: the first is suitable for operation on a shared memory architecture; the second provides garbage collection services as added functionality to a distributed shared virtual memory service.

1 Introduction

This paper presents a garbage collection technique suitable for multi-threaded applications running on multiprocessor targets with a shared memory abstraction. The shared memory abstraction can be provided by the architecture, or virtually, obtained through the Distributed Shared Virtual Memory paradigm [LH89][CBZ91]. The multi-threaded applications may be the result of compilation of programs written in programming languages with explicit or deduced parallelism. The different threads of control of the application are assumed to share a single address space, a global storage allocation and reclamation discipline is thus required.

These techniques were designed for implementing a parallel virtual machine and compiler for the language LCS [Ber88][BGG91], an extension of the language Standard ML [MTH90] with processes based upon the CCS formalism [Mil80]. However, most of the results are independent of this context; similar techniques would be required for parallel implementations of many programming languages with transparent memory management. In distributed implementations of LCS, the algorithms presented here are complemented by algorithms for distributed scheduling, load balancing, and a few other distributed functions which cannot be discussed here. A complete account of the experiment is presented in the first author's forthcoming thesis [LeS92].

The starting point of our technique is [ELA88]. The heap space is organized in pages; an incremental page based collection algorithm operates locally on each of the processors. Unlike Ellis et al., we reject the idea of having specialized collector or mutator threads. Instead, each thread here alternatively plays the role of mutator and collector. This allows a better cooperation of several threads for performing a single collection, when required, or working together to make the whole application progress. We can have at any time a varying number of collectors and/or mutators; the progress of collection versus allocation is controlled by a specific scan credit mechanism. The several processes share a virtual space of references, as in [Hal84]; but the assignment of storage to each processor is not statically determined. Instead, pages are dynamically allocated to the processors, for specific purposes. The page management algorithms guarantee the consistency of the information held in the pages.

Integrating the garbage collection mechanism with the other paradigms present in the underlying architecture received much attention. Two implementations of the allocation/reclamation algorithms are proposed. The first is suitable for shared memory services directly provided by physically shared memory architectures. A second targets systems with physically distributed memories. In this case, the page allocation and collection services are provided together with a virtually shared memory service, as added functionality, rather than running on top of it.

Section 2 reviews and discusses versus our goals a number of mechanisms taken from existing garbage collection techniques. The collection method is presented in section 3. Section 4 suggests a canonical implementation on shared memory architectures while section 5 investigates an implementation with a distributed shared virtual memory service. We conclude with some remarks prompted by a prototype implementation of the algorithms and discussion of some possible variants and enhancements.

2 Architecture

2.1 Copying collection techniques

Garbage collecting techniques have been developed for a long time; reference [Coh81] surveys a number of algorithms. A convenient terminology for discussing memory allocation and reclamation techniques is that of *mutators* and *collectors*. A thread is a mutator when it allocates new storage, or updates existing storage; it is typically the application to be run. A thread is a collector when the operations it performs are relevant to storage reclamation. The storage is assumed to be organized as a set of *cells*, possibly referring each others and residing in a *heap*. A number of references to such cells, called the *root*, constitute the context (or registers) of each mutating thread.

Among the methods based upon transitivity of references, the copying methods certainly are the most satisfying. They consist in copying the reachable cells from a first space called the *fromSpace*, into another space, called the *toSpace*. They yield fast collectors, since only the cells reachable from the root are visited; they are able to reclaim circular structures and they have the effect of compacting the heap, which reduces page thrashing. However, copying techniques generally yield a poor storage occupancy, but most of that space is not permanently required.

The basic stop and copy version of the technique suffers the essential drawback of requiring the mutator to stop while the collector is proceeding, resulting in relatively long latency. Baker's variant [Bak78] was aimed at solving this drawback. The mutator is resumed immediately after the cells referenced in the root data have been copied and their references updated in the root data. The cells copied are left unscanned. The mutator must only operate on cells in *toSpace*. For achieving this, each access from the mutator to a cell residing in *fromSpace* forces its copy into *toSpace*, if it was not copied yet, and updates its reference. A pointer, **B**, points to the next available address for copying cells; another pointer, **S**, points after the last cell scanned. New cells are allocated from a pointer **T**, initially set to the last available address of t*oSpace*, and decreasing towards the copy pointer **B**.

Conditions are established to make the whole mutator and collector process *real-time*, in a certain sense; Baker's algorithm is incremental. For each allocation of one word of storage, *k* words are scanned, where *k* is some non negative constant. For a program which has a maximum cell requirement of N words, with a half-heap size of t, the parameter *k* must be greater than N/(t-N) to guarantee that scanning is terminated before pointer **B** reaches **T** [Bak 78].

Baker's method is costly if the test it requires at each access of a cell for determining if it stands in *toSpace* or in *fromSpace* is done by software. Ellis et al. [ELA88] satisfactorily solved this drawback by using the memory protections facilities provided by the hardware to prevent access of unscanned cells by the mutator. The heap is organized in pages; to enforce the property that the mutator only sees scanned cells, all pages of the heap containing unscanned cells (i.e. between pointers **S** and **B**) are read-protected. An access to these pages by the mutator would be trapped by the hardware and an exception would be raised; the access exception is handled by scanning the faulty page and then relaxingits protection, allowing the mutator to resume.

2.2 Garbage collection for multi-threaded applications

We will only consider here parallel applications operating on a single virtual address space. The complexity of collection algorithms proposed for multiple address spaces applications results from the absence of an observable global state, and the problem of migrations of cells between the processors [Rud86].

The approach taken in [Hal84], also relying on a single virtual address space, is to have a global heap statically and equally partitioned, each sub-heap being under control of one of the (virtual) processors. Each processor executes on its sub-heap a collection algorithm based upon Baker's, with its *toSpace*, local roots, and using local pointer variables **T**, **S** and **B**. The changes brought to the basic algorithm for multiprocessor operation consist of a lock-bit associated with each address in *toSpace* and each cell in *fromSpace*, to ensure atomicity of the copy and update operations. A global synchronization ensures that all processors flip their spaces before one of them starts the next collection.

The Multilisp treatment suffers two major drawbacks. Firstly, garbage collection is initiated when any of the sub-*toSpace*s is full, whatever the content of the others, yielding in practice much more frequent garbage collections than would be required in the single-heap case. Secondly, each processor must copy for itself all the cells only locally referenced (the gray areas in figure 1 below), with no possible help from the other processes for scanning these.

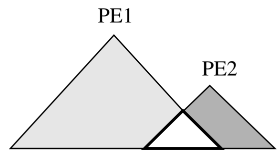

Figure 1. Sharing of heap cells

2.3 Architecture

The goal is to design the memory management layer for a virtual machine running LCS, an experimental parallel programming language. LCS programs, in which parallelism is made explicit, lend themselves naturally to parallel evaluation. However, typical LCS programs use a very large number of threads, from a few to several thousands, or tens of thousands, most of them very short lived. Consequently, the LCS virtual machine is constituted of a medium, fixed, number of processes, each handling execution of a number of LCS threads. A distributed scheduler, with a load balancing algorithm, distributes execution of the many LCS threads on the fewer processes.

The shared memory abstraction is adequate for our purpose. LCS, like many other high level languages, handles values which are generally trees, possibly with circularities. It would be very expensive to transmit copies of such values by messages between processes; LCS values are transmitted “by references”. Another argument for the shared memory abstraction is that typical LCS programs do not heavily rely on side-effects, yielding, in practice, relatively few write access conflicts on the shared heap.

From the previous review, we will retain the two-space copying and compacting method and the incremental technique [Bak78]. We will retain from [ELA88] the idea of using page protection traps for protecting parts of the heap, with a recovery mechanism. This, despite the added complication of a heap granularity distinct from the constituents of the heap (the cells), makes Baker’s method of incremental collection efficient. However, we will not retain the specialized mutator and collector threads advocated in this reference, and have instead threads which may be alternatively mutator or collector.

The page allocation mechanism allows processors to share the work of incremental scan; the progress of allocation versus collection will be controlled by a credit mechanism. While an incremental criterion similar to Baker’s is maintained, idling processes are requested to perform some scan until they acquire some work. Collection may then progress “in advance” over the mutation, and this will reduce the amount of scan to be done after future allocations. Using this mechanism, most of the potential advantages of collectors operating in parallel are recovered, while preventing the storage overflow problems which are difficult to avoid with asynchronous parallel collectors. In [ELA88], this particular problem is delayed, rather than avoided, by flipping spaces when occupancy of *toSpace* reaches some threshold, rather than when *toSpace* is full. Alternatively, [BSD91] prevents overflow by controlling the scheduling of the collector and mutator threads.

All processes share a single address space, but, conversely to [Hal84], the heap will not be statically partitioned over the different processors; we will make use instead of a paged heap, with pages dynamically allocated to any process. This treatment avoids the two major drawbacks of a statically partitioned heap; no process plays a particular role.

The collection technique is presented in the next three sections. The routines have been split as "front-end" routines, those interfaced with the application, and "back-end" routines, those directly using the storage services provided by the operating system.The front-end routines are discussed in the next section. The back-end routines may benefit from an encoding matching the underlying operating system services. Two implementations of the back-end routines are discussed. The first, in section 4, is suitable for any virtual shared memory target. The second, in section 5, provides these routines together with a distributed shared virtual memory service [LH89].

3 The multi-threaded garbage collector, front end routines

3.1 Storage services and layout of the storage of processes

The storage for each thread is constituted of a number of segments. Each segment is a contiguous address range partitioned in pages, all of the same size. A segment may be shared by several threads. We will assume that the pages which constitute a segment can be individually attached or detached from the address space of a thread, and that any access from a thread to a page not currently part of its address space raises an exception for which a handler may be set. These services are available on many stock operating systems; setting page protections at user level is a feature of Unix SVR4 and Mach based operating systems. Depending on the operating system, page protection facilities are either provided at thread level (otherwise sharing their address space), or at process level (possibly sharing part of their virtual space); we will use indifferently the words "thread", "process", "processor", or "site" in the following sections, to mean a thread of control associated with some shared storage for which page protections may be locally manipulated.

No assumptions are made about atomicity of updates on shared pages. The algorithms will use page locks where atomicity need be enforced. Any attempt to lock a page already locked should make the requesting process wait until the page is unlocked.

Each process will make use of three segments: two heap areas of identical size, referred to as *toSpace* and *fromSpace*, and a *stack* area. Each process shares with all others the heap areas. Though each process privately extends or shrinks its stack segment, it is convenient to have the stack segments shared by all processes so they can get help from the others for scanning their stacks. We took this choice here, though the alternative choice of having process stacks in private areas could have been taken as well, at the price of some extra complications for detecting end of collection. Finally, each process privately owns a set of registers and the three pointer variables, **S**, for scan, **B**, for copy, and **T**, for allocation of new cells; they also handle a number of private or shared auxiliary variables. To avoid contention on variables **S**, **B** and **T**, the processes will operate on different pages for each of these operations.

Pages are dynamically allocated to the processes, following a strategy to be made precise in a latter section, and released after use. The different pages may hold information of different nature (e.g. none, unscanned copies of cells, coherent cells, etc.), we will talk about the state of a page to mean the nature of the information it holds. The use of the storage, and the cooperation between the processes for achieving collection and allocations, are performed through of a number of page allocation and liberation routines, one for each “kind” of pages needed. These routines encapsulate the necessary synchronizations, and bookkeeping, for pages.

3.2 Heap allocation

A page will be identified by any address in the range it provides. New cells are allocated from a pointer **T**; the procedure `Allocate` is always called with a current **T** defined. The usual “end-of-heap” test is replaced here by an “end-of-page” test. When the cell (of size n) to be allocated does not fit on the current page, the current page is freed and as many pages as required for allocating this cell are requested by a call to `allocPageT`. The procedure provides pages for n contiguous addresses, attaches them to the address space of the process, and sets the local pointer **T**. If the required amount of pages cannot be allocated, then a flip is initiated. Procedure `fill` fills the space allocated and advances pointer **T**.

```
Allocate (n,content) =
        (if spaceLeftOnPage(T) < n
         then (freePageT();
                  Scan();
                  if not(allocPageT(n))
                  then (Flip();
                          if not(allocPageT(n)) then FAIL));
        fill(T,n,content));
```

In general, cells may overflow the page size. The solution proposed in [ELA88] is to associate with each page a “crossing” flag, set for pages not beginning with a new cell. The alternative taken here is slightly more space consuming, but allows to scan pages individually. It consists of associating with the crossing flag the minimum information needed to scan the part of the cell starting the page. This added information allows the scanner to see that part of cell as a complete cell; it would typically hold an indication of the nature of the cell, which determines a scanning method, and the size taken on the page. This information cannot stand in the pages themselves since it must not take any address slot. When crossing a page boundary, procedure `fill` releases the current **T** page by a call to `freePageT`, and fills the crossing information for the following page.

3.3 Incremental Scan

Each thread alternatively plays the role of mutator and collector. Following Baker’s incremental technique, each allocation of a word in the heap forces to scan a number *K* of words. The scan is performed after the page currently filled is released, rather than performed after each allocation; this reduces the overhead on allocations. A procedure `allocPageS` provides, and attaches, the next page to be scanned. It sets the scan pointer S, if some incremental scan is required, or, otherwise, returns false. Procedure `freePageS` releases page **S** when its scan is complete. This is repeated until a call to `allocPageS` returns false, meaning either that no more incremental scan is currently required, or that collection is complete. The pages are locked while being scanned, to prevent concurrent scanning by another processor (`allocPageS` locks, `freePageS` unlocks).

```
Scan () =
      while allocPageS()
      do (scanPage();
          freePageS());
```

Mutation continues after end of collection has been detected, until *toSpace* is full; this also contributes to decrease the average cost of allocations. The progress of scan versus allocations is controlled by a credit mechanism soon to be described.

3.4 Copying

Copies of cells resulting from scan are done on pages allocated by a third procedure, `allocPageB`. Copies are allocated from a pointer **B**. For allowing the scanner to work on a page basis, the cells are copied on pages distinct from those on which new cells are allocated. The procedure `allocPageB` locks the pages it allocates, attaches them to the address space of the thread, and sets pointer **B**. `freePageB` detaches the current **B** page, so that any further access to this page will provoke an access violation, and unlocks the page. The lock prevents other processors from attempting a scan of the page while it is being filled. In practice, `scanPage` would make use of the crossing information for pages, for determining the location and size of the first cell on the page.

```
scanPage () =
      if allocPageB(threshold)
      then (for all cell in page of S
            do scanCell(cell);
            freePageB())
      else FAIL;
```

All processes share, and have authorized access to, all pages of *fromSpace*. To prevent cells from being copied concurrently, the page on which an old cell stands is locked until the cell is copied, its copy-bit updated, and its forwarding address set.

```
scanCell(cell) =
      for all cell' referenced in cell
      do (lock(cell');
          if not(copied(cell')) then Copy (cell');
          update reference to cell' in cell;
          unlock(cell'));
```

The Copy routine copies cells from **B** and sets their copy-bit and forwarding address. The low level procedure copy advances **B**; it also releases the current **B** page and fills the crossing information of the following page when crossing a page boundary. If a cell does not fit on the current **B** page, then that page is freed and as many contiguous pages as required are requested by a call to allocPageB.

```
Copy (cell) =
        (if spaceLeftOnPage(B) < size(cell)
         then (freePageB();
                if not(allocPageB(size(cell))) then FAIL);
         forward=B;
         copy(cell,B);
         mark_copied(cell);
         set_forwarding_addr(cell,forward));
```

3.5 Scan on page access exception

Any attempt from a processor to access a page not part of its current address space raises an access exception. Upon an access exception, a handler routine, Handle, is invoked, which performs some treatment, and the computation is resumed at the instruction that caused the exception. There are a number of reasons for which a page accessed may not be part of the current address space of a process, including the case where the page has been filled with copies of cells, but has not been scanned yet. As in Ellis's algorithm, the mutator expects to see scanned cells only.

On that instance of access exception, the handling procedure first attaches the page to the address space of the process. The page is then locked to prevent concurrent scan by another process, and it is scanned. Upon completion of scan, the page is freed and unlocked (by a call to freePageS). Conversely to [ELA88], all page protections here can be manipulated at user level.

From now on, that particular page holds cells with consistent references, and no exception will be raised when accessing it from that process. But the page is still non-attached by the other processes. It is assumed that, given a page of *toSpace*, a process can decide if this page was scanned or not. The handling routine, which is the same for all processes, should first check that the faulty page has not been scanned yet, before scanning it; otherwise it just attaches the page.

3.6 Stack allocation and scanning

Stacks are the most usual technique for implementing parameter passing. If application stacks were encoded in the heap, with stack frames encoded as cells, then only the stack pointer would be part of the root references of a process. But, essentially for efficiency reasons, function call stacks are often encoded as arrays, in a dedicated area. It is this entire array which must then be considered root data by the collector. Moreover, stacks may grow very large on languages favoring recursion, such as LCS; scanning the full stacks at flip time, as should be done for all root references, might break the low latency requirement for the mutation process, as noticed in [Bak78] and [ELA88].

Stack frames are allocated on contiguous pages in an area distinct from the heap. After a flip, all the pages below the topmost stack page are detached from the address space of the thread. Only the topmost page is scanned as part of the root data. The other pages will be scanned on access exceptions or by incremental scan. This strategy implies that stack pages may be scanned individually, without knowledge of the boundaries of stack frames. This constraint is easily satisfied if multi-words values are allocated not in the stack area, but in the heap instead, at the price of an extra indirection. The stack should only hold references to heap cells and constant values, which must be encoded as if they were occurring in a heap cell.

On an access exception on a stack page, the handler routine scans the faulty stack page and the mutator is resumed. Stack pages are scanned by the previous `scanPage` routine (we may assume a predefined crossing information, identical for all stack pages), only words at addresses below the stack pointer are scanned.

As for the heap, the stacks must be incrementally scanned. A number k' of stack words must be scanned for each word allocated, so that the stacks, of total size s, are scanned before *toSpace* is filled. Parameter k' depends upon the size s of the stacks, known at flip-time. The smallest number of words allocated between two flips is $t/(k+1)$, where t is the size of *toSpace* and k is Baker's parameter for the heap. Allocating that amount of words should force a complete scan of the stacks; consequently: $k' = (k+1)*s/t$[1].

Practically, the words in the stacks and those in the heap are globally considered, and $K=k+k'$ words are scanned for each word allocated. Parameter K is adjusted at flip time, from a fixed k and a parameter k' computed from the previous equation. The routine `allocPageS` allocates either an heap page or a stack page, as long as some is available. Furthermore, since we choose to have the stack areas shared by all processes, any process is able to scan the stack of any other. Finally, in order to favor cells which have the longest life expectancy, the stack pages are allocated by `allocPageS` from bottom to top.

3.7 Idling processes and the scan credit

Processors idling by lack or work are requested, through a `background` procedure, to perform some scan until they get some work (by a scheduling and load balancing mechanism not discussed here). The words scanned from the `background` procedure or upon access exceptions are considered as words scanned "in advance" for allocations to come (a credit on the amount of words to be incrementally scanned).

In order to fulfil the low latency requirement, the credit mechanism is implemented with two credits: A *local* credit (one per thread), decremented by `freePageT` (of the amount of words allocated on the page, multiplied by constant K) and incremented after scanning a page

---

1. Baker uses $k' = k*s/n$, where n is the amount of words in the heap when collection is complete. The difference comes here from the fact that flip does not occur at end of collection, but when *toSpace* if full. n is unknown at flip-time, then, but an upper bound is deducible from k and t, by $n = t*k/(k+1)$.

(upon an access exception handling or a local incremental scan), and a *shared* credit, incremented by `freePageS` when it is invoked from the `background` procedure.

When allocating, a process uses a maximum amount of its local credit. If the process has to do some scan (i.e. if its local credit is negative), then it first checks the shared credit. If the shared credit cover its debt, then no scan is performed and the shared credit is decremented of the local debt. Otherwise, the local debt is decreased of the shared credit, the shared credit is reset and scan proceeds. Finally, when a processor is idling, it first transfers its local credit to the shared credit, so that other processes can benefit of it, before scanning for the other processors.These computations, as well as the test deciding if incremental scan is required, are encapsulated in the procedures `freePageT`, `freePageS` and `allocPageS`.

The scan credit mechanism brings two main advantages. Locally, it makes incremental scan lazier; the amount of words scanned on access traps or by idling processors are subtracted from the amount of words to be incrementally scanned. Globally, It allows the processors to cooperate for scanning, some of them performing part of the scan task for the others, with an actual parallelism. The number of collectors is dynamic and the algorithm controls the progress of the collectors versus the amount of data allocated.

3.8 Flip

A thread notices that *toSpace* is full when a call of `allocPageT` returns false; it then initiates a flip. A synchronization of all threads is necessary before the local flips take place, to prevent the different threads from working with different views of the heap. Then, all threads stop mutation, release the pages they held, and start a local `flip` procedure. It must be noticed that, when a flip occurs, the previous collection is necessarily complete; all threads are then either idle or mutating.

The `flip` procedure itself first consists of inverting the roles of the *to* and *from* spaces. Each thread then attaches to its address space all the pages of *fromSpace* (the former *toSpace*) and detaches all the pages of *toSpace*. The pages of *toSpace* will be attached on request. All stack pages are also detached, except the topmost page. The thread then initializes its local bookkeeping information and variables (including a **T** page for non-initiators), and scans its registers and top stack page. Mutation is subsequently resumed.

The initial attachment of the pages of *fromSpace* avoids nested access exception handling. The initial detachment of all pages of *toSpace* is the consequence of the strategy retained for page management. This strategy is to force an access exception in a process every time this process has no precise knowledge of the content of a page; the access exception handler will determine the exact nature of the contents of the faulty page, and perform the adequate treatment. To avoid detachment of pages attached in several processes, the page management guarantees that, once a page is made available to a mutator, it will never need to be detached from any process until the next flip.

3.9 States of pages, and transitions

Pages have states, identifying the nature of their contents. Initially, and just after a flip, the pages of *fromSpace* have no significant state and are attached in all processes. The *stack* pages have one among two states: Scanned, or Unscanned. The pages of *toSpace* are initially detached from all processes and all of them initially have state Empty. The figure below depicts the different possible states and transitions of a page in *toSpace*, with meanings obvious from the previous sections.

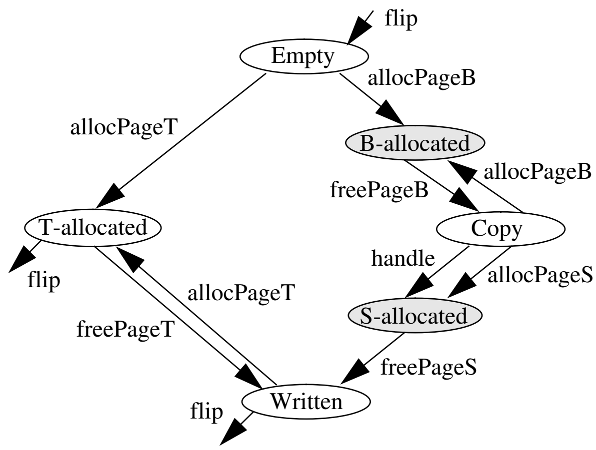

Figure 2. The different states of a page in *toSpace*

Consistency of the information held by the pages requires that the pages in gray states in figure 2 are locked by the processor which set them in that state; they must not be available to the other processors until they leave that state.

Pages may be released by `freePageB`, `freePageT` or `freePageS` even if not completely filled. In order to maintain a reasonable mean page occupancy of the storage, a policy of reallocating partially filled pages may be adopted. Given a threshold `h`, a page will be said full if less than `h` address slots are available on that page. Partially filled pages in state `Copy` will be reallocated by `allocPageB`, partially filled pages in state `Written` will be reallocated by `allocPageT` (when the size requested is not greater than threshold `h`, in both cases). This explains some of the transitions in the above graph.

4 A canonical implementation of the back-end routines

4.1 Page pools

Pages must be allocated according to their state. A convenient encoding, easing retrieval of pages, is to store addresses identifying these pages in different data structures, according to their states. We shall need five, shared, pools of pages:

The EMPTY pool holds pages which have never been allocated. If empty pages are chosen to be the consecutive pages following the last allocated page in the *toSpace* segment, then it is not necessary to actually record their addresses; a shared page pointer protected by a semaphore is sufficient, incremented of the page size after each allocation of an empty page.

The WRITTEN pool holds the partially filled pages in state Written. Totally filled pages in state Written need not be recorded since they will never be reallocated. A semaphore protecting a linked list of page addresses is a suitable encoding.

The next three pools hold pages in state Copy. The STACK pool holds all unscanned stack pages (which can be thought of as pages in state Copy); the COPYP pool holds partially filled pages and the COPY pool holds full pages. Implementation of the access exception handler and of the page allocation routines will be made easier if, first, we have the ability of removing a page identified by its address from these pools, and, second, if the three pools are protected as a whole from concurrent updates. These pools may be represented by double-linked lists, and a semaphore, requested by `Handle`, `allocPageS`, `allocPageB` and `freePageB` should protect access to all three pools.

In the handler routine for access exception, it is necessary to check if a page is in state Copy, before scanning it. To avoid scanning the pools then, it is convenient to keep that information in a page table. This page table must also record some additional bookkeeping information which is summarized now.

4.2 The page table

The page table must be shared by all processes. What need to be recorded for operation of the algorithms is a Scanned/Unscanned bit for each stack page, a lock bit for each heap page, plus, for each page in *toSpace*:

- A Copy/Written bit.
- A crossing information. Its minimum size is that of a few bits specifying a scanning method, plus the number of bits for recording at most the page size (in words).
- The next available address slot on the page (for reallocations, scan, etc.); need to be large enough to hold the page size (in words).
- Room for encoding the pools discussed above as linked lists.

Let us assume that each page holds 4K bytes and that *toSpace* holds at most 16M bytes (that should cover the needs for most applications). Then 64 bits per page of *toSpace* are largely sufficient for storing the bookkeeping information. That amounts to 0.2 percent of *toSpace*; a fairly reasonable amount. In addition, a bit table, with one bit per page, is required for the stack area.

4.3 Page allocators and protection violation handler

Procedure `allocPageT` will first try to reuse a partially filled page from pool WRITTEN, if any is available and if the size requested is not greater than threshold `h`. Otherwise, it will attempt to allocate a range of pages from pool EMPTY. `freePageT` sets the state of page **T** to `Written` and, if the page is only partially filled, records it in the pool WRITTEN. The other effects of `allocPageT` and `freePageT` have been discussed in section 3.

Procedure `allocPageS` repeatedly tries to find an unscanned page until either one is available, or no more scan is required, either because end of collection has been detected, or because no incremental scan is currently required. In each attempt, it first tries to allocate an unscanned stack page (from bottom to top) from the STACK pool, then it attempts to use a page from pool COPY, and finally a page from pool COPYP. While searching for a page in these pools, it requests the semaphore protecting these pools. Procedure `freePageS` sets the state of page **S** to `Written`, and saves it into the pool WRITTEN if partially filled.

`allocPageS` is also in charge of detecting end of collection. A shared `Unscanned` counter is maintained to detect end of collection. This counter records the number of pages allocated for copies of cells which are still unscanned. This counter is initially set to the number of unscanned stack pages after flip; it is incremented every time a page is taken from EMPTY by `allocPageB`, and decremented every time a page is released by `freePageS`. Collection is complete when this counter reaches 0. Obviously, this counter must be protected from concurrent updates.

The scan credit mechanism is easily implemented with two counters, one being shared and protected by a semaphore. Both credits are initially 0, and reset to 0 by flip. The local credit is decremented by `freePageT` of the amount of words allocated on the page being freed since it was last allocated (multiplied by the incremental scan parameter K), and incremented by `freePageS` of the amount of words on the page freed. The shared credit is incremented by `freePageS` when called from the `background` procedure invoked when processors are idling.

Procedure `allocPageB` first checks if the size requested fits into the space available on the page **S** being scanned. If this is the case, then it is this page which is also allocated for copying, and **B** is set to the first available address on that page. Otherwise, `allocPageB` tries to allocate either a partially filled page in state `Copy`, from pool COPYP, if any, or a range of pages from pool EMPTY. If pages **B** and **S** are distinct, then procedure `freePageB` detaches the current **B** page, sets its state to Copy, stores it in the COPY pool (if full), or COPYP pool (if partially filled), and finally unlocks it.

The handler for access exceptions must request the semaphore protecting the three Copy pools before attempting to scan the faulty page. It then locks the page and checks if its Copy bit is still set. If this is the case, it removes the page from the pool it lied into. The Copy pool semaphore is then released; the page is scanned, if required, and is unlocked.

4.4 Flip

All processes must flip synchronously. There may be several processes simultaneously noticing the necessity of a flip; multiple simultaneous flip initiations must be prevented. It may also happen that some processes do not locally notice the necessity of a flip (e.g. because these processes currently do not allocate data in the heap); so a mechanism is needed to make all processes aware of the necessity of a flip.

Selection of an initiator is easily implemented with the help of a semaphore. Among the potential initiators, the first that could acquire the resource broadcasts a signal to the other processors and waits for all of them to reply. It then releases the semaphore and broadcasts another signal, enabling local flips on all processors. This solution is basically similar to that used for Mul-T [KHM89].

A number of variables must also be initialized at flip time. These include the `Unscanned` counter, the shared and local credit counters, the current pointer **T** (needs to be allocated for non-initiators; the initiator does it from the `Allocate` procedure), and the incremental scan parameter K, which must be adjusted after each flip since it depends upon the total stack sizes at flip time. We can assume that the initiator takes care of initialization of the shared counters, in addition to the initialization of its local data.

5 Providing Garbage Collection with a Distributed Shared Virtual Memory service

5.1 Goals

This section addresses the implementation of the algorithm on distributed memory architectures. The concept of Distributed Shared Virtual Memory (DSVM for short) [LH89][CBZ91] provides a shared memory abstraction for a physically distributed memory architecture. A first approach would be to implement the algorithms discussed in sections 3 and 4 on top of a DSVM service, since it supplies the required abstraction (assuming some implementation of semaphores is available).

An alternative, hopefully yielding better performances by reducing the page traffic, is to provide a garbage collection service as an added functionality to a DSVM service, rather than built on top of it. This section specifically investigates this issue. The abstract procedures discussed in section 3 need not be altered, only the implementation of back-end routines should. Page management for the allocation and garbage collection service will be integrated, in some sense, with the page management required for providing the DSVM service.

We will assume our application to run on a number of (virtual) processors, or "sites". All processors virtually share a single address space through a DSVM service. As before, each processor owns part of the roots of the application and shares with the others all cells in the heap, and possibly its stack pages. The management of control information will rely here on message passing.

5.2 Page managers

The allocation/collection algorithms require to maintain some bookkeeping information for pages (crossing information, state information, etc.). These information cannot be saved within the pages, since that would break the required continuity of address space across pages. The solution taken is to partition the management of state information for pages between the processors, the local bookkeeping information being kept in some specific area on each processor. We will further assume that each processor is statically assigned a range of pages to manage, and that each processor is aware of the distribution of page management.

This strategy is exactly Kai Li's Fixed Distributed Manager strategy for providing the distributed shared memory service [LH89]. Upon receiving an access request for a page it manages, a processor takes the adequate decision, according to the state of that particular page, and then, once access is enabled for that page, the page is transferred to the requester.

5.3 Allocation of pages for new cells

Locally, the processors must organize the information concerning the pages they handle; the method proposed in section 4, with a page table and page pools, is still adequate here. In order to minimize the communication traffic, the processors preferably allocate for their own usage the pages they manage. In order to allow all pages to be allocated, the processors that manage pages that may be allocated for storing new cells (or free pages, for short), are organized as a virtual ring; a distributed algorithm takes care of allocation of these pages to the whole set of processors. Initially, all sites are in the ring, and the successor of each site is given by some predefined ordering known by all sites.

Every site records the remote site that replied to its last request for a free page, initially set to the next site in the ring; this site is referred to as its *remote free page allocator* in the sequel. A site receiving an allocation request for a free page will handle it as if it was a local request. If the request can be satisfied locally, then a range of addresses is returned to the requester. Otherwise, two cases may occur: either the request may still be satisfied by another site in the ring, in this case the processor transmits the request to its own *remote free page allocator*; or no site in the ring may satisfy the request, in this case, a failure message is sent to the requester, which will initiate a flip upon reception of the message.

If a demand is satisfiable, then it must be satisfied before it realizes a complete turn of the ring of sites managing free pages. Any site not managing any free page will never manage any in the future; sites may only leave the ring. The condition is checked as follows: each site visited by a request compares the identifier of the requesting site (say i) with its own identifier (say j) and that of its *remote free page allocator* (say k). The previous condition, and the way remote free page allocators are maintained, imply that no site between j and k (in the initial ring) manages free pages, thus, the request cannot be satisfied if the requester stands between j and k (in this ordering).

Upon a successful reply to its request for free pages, a processor updates its *remote free page allocator* to be the site that satisfied its request, except in the infrequent case of a multi-pages request. In this last case, the *remote free page allocator* is not updated since the request may have visited sites that manage free pages, though none of them could cover its demand.

Freeing a page in state Written consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in sections 3 and 4.

5.4 Allocation of pages for scan

For incremental scan, the processors must be able to allocate the whole set of pages that require scan. Conversely to the case of free pages, or of pages for copies, it is not sufficient here to organize the sites managing pages to be scanned as a ring. The problem is that the number of pages to be scanned managed on a site does not monotonically decrease. As long as a processor manages pages in state Empty, these may be allocated for copying cells and consequently become pages to be scanned. Using a particular allocation strategy for allocating pages for copies, and with some additional constraints on cell sizes, the ring structure could still be used, as shown in [LB91], but these constraints are not assumed here.

The sites managing pages to be scanned will be organized as a virtual distributed queue. Initially, all sites agree on the first and the last site in the queue (these can be statically determined, or dynamically, for instance at flip time). The algorithms guarantee that any request for a page to be scanned either will reach the first site in the queue, which will reply to the request, or, if the queue is empty, will reach a site that detected end of collection, which will propagate this information to the requester.

An implementation of the algorithm is feasible using four variables per site. Each site records the last site having provided to it a page to be scanned, that's its *remote allocator for pages to scan*. In addition, each site in the queue maintains its *successor* site in the queue (initially the next site according to some predefined ordering, or itself if the site is the last in the queue). Each site also maintains a variable holding the site it believes to be the last in the queue (the *last* for short), and, finally, a flag indicating if the site has detected end of collection. The insertion and removal protocols for maintaining the queue guarantee that, by transitivity of the *remote allocator for pages to scan* references (resp. *last* references), the first (resp. the last) site in the queue is actually reached. Further, they guarantee that the *successor* variable on each site either holds an indication that the site is not in the queue, or, if that site is in the queue, holds its successor in the queue (itself if it is the last in the queue). If the queue is empty, then the last known site in the queue necessarily detected end of collection, and has its specific flag set.

For the same reasons than for allocation of free pages, each site preferably allocates for itself the pages to be scanned that it manages. To help detection of end of collection, each site locally maintains an `Unscanned` counter, handled similarly to the `Unscanned` counter discussed in section 4, but relative to the pages it manages. This counter maintains the number of pages allocated for copies (local or remote) the scan of which did not terminate yet. A request for a page to be scanned (local or remote) is delayed by a site until either a page to be scanned is locally available, or the `Unscanned` counter reaches zero.

When a request cannot be locally satisfied, the site, if not in the queue or if not the first in the queue, transmits the request to its own *remote allocator for pages to scan*, which handles the request as if it was locally issued. Any site visited which is aware of end of collection replies negatively to the requester, making it aware of end of collection. If the receiver of a request is currently the first in the queue, then, either it currently manages some pages to be scanned, in which case it sends one to the requester, or it does not manage any of them, in which case it leaves the queue and then transmits the request to its *successor* in the queue. Furthermore, if the receiving site is also the last in the queue and does not manage any page to be scanned, then collection is complete; the site sets its end-of-collection flag, replies negatively to the request, making the requester aware of end of collection, and leaves the queue (which becomes empty). A site enters the queue, following the last in the queue, when not currently in the queue and acquiring the management of a page holding copies (see 5.5).

This solution may appear rather "centralized"; a queue is actually the most natural structure here, due to the non-monotonicity of the number of pages to be scanned on each site. In addition, it should be noted that the solution proposed realizes a balancing of pages to be scanned over the sites. Acquiring a page to be scanned from the queue may lead to copy some cells which, as long as the requesting site can provide pages for these copies, will be locally copied. This treatment has thus the potential effect of locally creating pages to be scanned.

Freeing a scanned page consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in section 3 and 4.

5.5 Allocation of pages for copies

The allocation of pages for copies uses an algorithm similar to the one used for allocation of free pages. However, it is necessary that these two virtual rings are distinct, since the local allocation strategies for free pages and pages for copies differ (for copies, one first tries to allocate a partially filled page in state Copy, rather than Written). Each site maintains a *remote allocator for pages for copies*, managed similarly to the *remote allocator for free pages*.

The algorithm for detection of end of collection assumes that collection is complete if and only if all local `Unscanned` counters hold the value 0. In order to maintain the property, a site able to reply to a request for a page for copies, for itself or for another site, must join the queue of sites managing pages to be scanned (if not in the queue yet) before returning the required range of pages to the requester.

Feeing a page holding copies consists of requesting the manager of the page to perform a local release of the page. Locally, one proceeds as in section 3 and 4.

5.6 Page locks and page protection violations

The pages in states S-allocated or B-allocated, as well as the pages holding cells being copied, must be locked to enforce consistency of their content. Instead of using a lock bit here, as was done in section 4, it is more efficient, considering that the shared memory is obtained through exchanges of pages, to request these pages to their manager with exclusive write access. The protection will be relaxed when the page is freed by the processor to which it was granted. It is the manager of the page which will enforce the exclusive access and make all requests for that page wait until the page is freed. Implementation of this primitive should not require any additional effort, since it is also required for implementing the DSVM service.

When an access violation is trapped by a processor, it must not necessarily scan the page. If the same distribution of the page management task is taken for both the DSVM service and the collection service, then handling access traps in the mutation process does not require any additional message. In both cases, it is enough, when trapping a protection violation, to ask the manager of the page for the action to be taken.

5.7 Distributing the scan credit mechanism

As in section 4, a local credit counter is maintained by each processor. The shared credit counter is here distributed over the processors; the actual shared credit is the sum of these locally maintained shared credits. The following discipline is adopted for maintaining the shared credit counters:

- When an idle processor scans for another, it decrements the remote shared counter.
- An idle processor first transfers its credits (local and shared) with its first request for a page to be scanned, since the credits cannot be used locally.

- When using the shared credit, a processor does not use the full available credit (which is distributed over the processors), but uses the shared credits of the processors it visited for finding a free page, and, if scanning is still necessary, the shared credit of all processors visited for finding a page to be scanned.

With this method, the distribution of the shared credit mechanism does not require any additional message.

5.8 Flip

The necessary synchronization of all processors preceding the flip requires a distributed synchronization algorithm. This algorithm must prevent from multiple, simultaneous, flips, by electing an initiator among the sites that noticed the necessity of flip, and must ensure that all sites flip. An ad hoc algorithm, requiring a pause on each processor corresponding to the time necessary for transmitting four messages, is discussed in [LeS92], to which the reader is redirected for details.

6 Discussion and Experiments

6.1 Enhancements

We assumed so far a fixed number of threads, each alternatively acting as a mutator or as a collector. Allowing a varying number of threads, instead of a fixed number, may be convenient for implementation of programming languages in which programs typically use fewer threads than LCS, with longer life expectancy. This may also be convenient for dynamically adapting the execution of LCS programs on the parallel virtual machine to the physical resources available. Augmenting, or reducing, the number of threads in the shared memory case (section 4) does not imply a large work, the main task for an entering process is to acquire a local context from one of the other threads; similarly, a leaving process must transmit its root data to one of the remaining processes. This can be achieved through some load balancing mechanism. In the distributed case, in addition, the storage managed by a leaving process has to be redistributed over the other processes, and an entering process must acquire from the others some storage to manage, unless the storage is simultaneously updated. In any case, all sites must be made aware of introductions or deletions of sites, and a reconfiguration of the page tables and distributed algorithms is required.

Dynamic extension of the heap size and/or stack size is another desirable enhancement of the basic algorithms.

To avoid having to move the contents of the heap, heap size adjustment would typically occur at flip-time, before the local flips take place. At that time, the content of *fromSpace* is irrelevant and that area can be replaced by another, of the required size, possibly at another place in the virtual space. Just before the next flip, the current *fromSpace* will be adjusted accordingly, to match the size of the current *toSpace* (or according to some heuristics for heap size dynamic adjustment). Besides moving an (empty) area, the adjustment also consists of updating the page tables accordingly.

Dynamically extending the stack should not be difficult too, provided the stack does not hold any reference to itself (that hypothesis was assumed in section 3). If the stack segment cannot be grown in place, then it must be detached and reattached at a place where it can be grown. As for the heap, an update of the page tables is required.

An enhancement of heap allocation that does not appear feasible at that time, unless taking strong limitations on the size of cells, is to use page protection traps for avoiding the end-of-page test required in routine `Allocate`, similarly to the solution proposed in [App88]. The problem here is that the heap may contain interleaved attached and detached pages, or ranges of pages. Consequently, the fact that some address is enabled (resp. not enabled) at a given distance from pointer **T** in some process does not imply that all addresses in between are enabled (resp. not enabled).

Other desirable enhancements include implementing a generational collector on this ground. This has to be investigated, but we cannot foresee any major reasons preventing from adding generational capabilities to our collector.

6.2 Experiments

The incremental scan, plus the use of page protection traps for enforcing the scan of pages holding copies, should provide a low latency collection mechanism with an acceptable loss of performances compared to a stop and copy collector. This has been confirmed by the experiments.

A version of the (sequential) LCS virtual machine equipped with an incremental collector based upon the algorithms given in section 3 was prototyped, and its performances were compared with those of the currently available implementation of LCS, which uses a stop and copy-depth-first collector, and with those of a version using a stop and copy-bread-first collector. The different versions were compared on various benchmarks; the table in figure 3 below shows the results for a benchmark consisting of running an implementation of the Knuth-Bendix rewrite-rule completion algorithm on an example set of rules. All collection algorithms were implemented in C; all were run with a heap-size of 16MB (8MB per space); all runs required 58 flips.

	s/c b-first	s/c d-first	incr 8kB	incr 64kB
Collection time/Total time (%)	19.7	26.3	24.3	22.1
Relative mutation time	1	0.97	1.14	1.11
Relative collection time	1	1.42	1.50	1.28
Relative total time	1	1.07	1.21	1.14

Figure 3. Performances of stop and copy and incremental versions

For this benchmark, an overhead of 21% in total mutation+collection time was observed for the incremental version, compared to the stop and copy-breadth-first version. For reasons which will not be detailed here, the current implementations of LCS use a depth-first variant of the stop-and-copy collection algorithm. The depth-first version is slightly faster than the breadth-first version, for mutation, but significantly slower for collection (the algorithm uses a pointer-reversal technique for achieving a depth-first copy of the active cells). Compared to the latter version, the overhead of the incremental algorithm decreases to 13% in total time.

The 21% overhead observed for the incremental version, versus the stop and copy-breadth-first version may seem high, compared to the 4% overhead claimed in [ELA88]; this figure requires some comments.

- First, the stop-and-copy versions have been used for several years and have been carefully optimized, which was not the case for the incremental version. The overhead should be slightly reduced by a careful optimization of the incremental version.
- The Collection time/Total time ratio observed for this benchmark is rather high; it was more often below 10% in the other benchmarks we tried. Considering that the overhead is greater for collection than for mutation, the typical total overhead should be lower.
- Another factor that influences the overhead is the ratio between words scanned on access faults and those scanned by incremental scan; again, this ratio was rather high here. Not surprisingly, it has been observed that the overhead decreases with that ratio. This stresses the need for a careful implementation of the access handler routine and of the page protection mechanisms, both at the application level and at operating system level.
- Finally, we exercised the incremental version for several page sizes, ranging from 8kB to 64kB. The smaller number of access faults on heap pages resulting from larger pages makes the overhead decrease when the page size increases. But, obviously, using large pages affects responsiveness of the applications.

As a conclusion, the typical overhead of the sequential incremental version, compared to the fastest stop-and-copy version, can be expected to be around 10%. This may be considered an acceptable overhead, considering the benefits of the incremental version with respect to responsiveness.

The scan credit mechanism, and the global page allocation, should allow processes to effectively cooperate for both collection and mutation. A simplified version of the parallel collection algorithm has been prototyped to run on a stock workstation running an SVR4 based operating system, and integrated in a preliminary version of the parallel virtual machine for LCS. The virtual machine is constituted of a number of processes (typically four to sixteen), running on distinct processors when allowed by the hardware. The effect of lazy scanning of the heap due to the credit mechanism could be precisely observed: processes lacking of work (including those waiting for I/O operations to complete) perform some scan for themselves, and then for the other processes. This is particularly interesting for interactive applications; most of the scan work is then done while the user is typing commands, with less overhead on the computations themselves.

Unfortunately, no fair performance figures can be provided yet for the parallel implementation. The task of properly integrating the memory management layer with the other components of the virtual machine is still in progress. However, an additional overhead is to be expected, compared to the sequential incremental version, due to the locks required for enforcing consistency of pages, in the shared memory version, or to the latency of page transfers, in the distributed version. With several processors running a single-threaded application, we should recover the advantages of parallel collectors; for multi-threaded applications, there should be an overhead on total mutation time, compared to the same application run on a single processor, but we should also observe an improvement in elapsed time.

References

[App 88] A. W. Appel, Simple Generational Garbage Collection and Fast Allocation, *Software Practice and Experience* 19(2):171--183, February 1988

[Bak 78] H. G. Baker, Jr., List Processing in Real Time on a Serial Computer, *Communications of the ACM*, 21(4), April 1978

[Ber 88] B. Berthomieu, LCS: une implantation de CCS, In A. Arnold, editor, *Troisième colloque C-cube*, Angoulème, France, Décembre 1988

[BDS 91] H-J. Boehm, A. J. Demers, and S. Shenker, Mostly Parallel Garbage Collection, In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989.

[BGG 91] B. Berthomieu, D. Giralt, and J.-P. Gouyon, LCS Users Manual, LAAS Report 91226, CNRS-LAAS, June 1991

[CBZ 91] J. B. Carter, J. K. Bennet, and W. Zwaenepoel, Implementation and Performance of Munin, In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, October 1991.

[Coh 81] J. Cohen, Garbage Collection of Linked Data Structures, *Computing Surveys*, 13(3), September 1981

[ELA 88] J. R. Ellis, K. Li, and A. W. Appel, Real-time Concurrent Collection on Stock Multiprocessors, In *SIGPLAN'88 Conference on Programming Language Design and Implementation*, June 1988. Also Digital SRC Research Report number 25

[Hal 84] R. H. Halstead Jr., Implementation of Multilisp: Lisp on a Multiprocessor, In 1984 *ACM Symposium on LISP and Functional Programming*, August 1984

[KHM 89] D. A. Kranz, R. H. Halstead Jr., and E. Mohr, Mul-T: A High-Performance Parallel Lisp, In *SIGPLAN'89 Conference on Programming Language Design and Implementation*, June 1989

[LB 91] T. Le Sergent, and B. Berthomieu, Un ramasse miettes distribué incrémental sur une mémoire virtuelle partagée distribuée, LAAS Report 91373, CNRS-LAAS, Novembre 1991.

[LeS 92] T. Le Sergent, Méthodes d'exécution, et machines virtuelles parallèles, pour l'implantation distribuée du langage de programmation LCS, PhD. thesis, 1992 (forthcoming).

[LH 89] K. Li, and P. Hudak, Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems*, 7(4):321--359, November 1989

[Mil 80] R. Milner, *A Calculus of Communicating System*, LNCS volume 64, Springer-Verlag, 1980

[MTH 90] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990

[Rud 86] M. Rudalics, Distributed Copying Garbage Collection, In *1986 ACM Conference on LISP and Functional Programming*, August 1986.