A Localized Tracing Scheme applied to Garbage Collection

Yannis Chicha* and Stephen M. Watt

Department of Computer Science
University of Western Ontario
London Canada N6A 5B7

Abstract. We present a method to visit all nodes in a forest of data structures while taking into account object placement. We call the technique a *Localized Tracing Scheme* as it improves locality of reference during object tracing activity. The method organizes the heap into regions and uses trace queues to defer and group tracing of remote objects. The principle of localized tracing reduces memory traffic and can be used as an optimization to improve performance at several levels of the memory hierarchy. The method is applicable to a wide range of uniprocessor garbage collection algorithms as well as to shared memory multiprocessor collectors. Experiments with a mark-and-sweep collector show performance improvements up to 75% at the virtual memory level.

1 Introduction

Many algorithms require visiting all objects in a forest of data structures in a systematic manner. When there are no algorithmic constraints on the visiting order, we are free to choose any strategy to optimize system performance. This paper examines an optimization of object tracing to improve performance in a memory hierarchy. The basic idea is to delay the tracing of non-local objects and to handle them all together on a region-by-region basis. We call this a *localized tracing scheme* (LTS).

An LTS organizes its visit of the heap based partly on the graph of objects and partly on the location of objects. A consequence is that LTS can be memory hierarchy friendly, which means we are able to optimize visits of objects at different levels of the memory hierarchy, including cache, virtual memory and network. At one level, LTS can be used to reduce paging and keep object tracing in main memory as much as possible. At another level, as on-chip cache memory increases in size, LTS may be used to minimize traffic between cache and main memory. LTS may also be used in modern portable devices, where relatively large and slow flash memory cards extend the smaller and faster device memory.

Our LTS technique is based on dividing the heap into *regions* with a *trace queue* associated to each region to hold a list of objects to visit. Trace queues are the origin of the performance improvements displayed by the LTS. They are

* Present address: Teradyne SAS, 3 chemin de la Dhuy, 38240 Meylan, France.

used to delay the tracing of remote objects, allowing tracing to concentrate on local objects. This enhances locality of reference by relying on object location, rather than object connectivity, to order tracing. The sizes of regions and trace queues are determined by the level of the memory hierarchy that we wish to optimize. For example, to obtain a cache-conscious algorithm, a region and the trace queues should be small enough to fit entirely in cache.

This idea may be applied to memory management, where reachable objects must be visited as part of garbage collection. Uniprocessor garbage collection is mature and offers satisfactory performance for many applications. In fact, garbage collection is now an integral part of the run-time for popular programming languages, such as Java and C#, which serve as the delivery platform for widely used applications. Improvements in garbage collection technology can therefore have impact on a broad user base.

We note that adding LTS to an existing collector is a relatively easy operation. This consideration is important in practice, where vendors are reluctant to make significant modifications to products' memory management for fear of introducing bugs. We have found it to be straightforward to modify two garbage collectors to use LTS, that of the Aldor programming language [1, 2] run-time support and that of the Maple computer algebra system [3].

The impact of localizing tracing depends on the garbage collection method in use: Mark-and-sweep collectors first visit all live objects, marking them, and then sweep the memory area to recover unused space. Optimization of memory traffic during the *sweep* phase has been considered by Boehm [4]. We observe that memory hierarchy traffic can also be improved during the *mark* phase using LTS. Since objects do not move in memory with mark-and-sweep, the benefits of LTS are similar at each GC occurrence. Improvements of the overall GC time decrease when few objects are live. In this case, the mark phase is short and optimizations have a small impact.

Stop-and-copy garbage collectors can move objects to new locations at each GC occurrence. To do this, they must visit all live objects. Generational collectors [5] also use tracing because each generation is handled by a copying or mark-and-sweep collection algorithm. In these cases the tracing may be performed using LTS.

The rest of this paper is organized as follows. Section 2 describes a family of localized tracing algorithms. Section 3 gives an example to illustrate the LTS. Section 4 presents an informal proof of correctness for this algorithm. Section 5 details our experiments and results with the GC for the Aldor run-time environment. Section 6 explores advantages and drawbacks of the LTS in a multiprocessor environment. Section 7 discusses related work in various garbage collection settings. Section 8 suggests directions for future work and concludes the paper.

2 The Localized Tracing Scheme

2.1 Depth-First Tracing

We start by considering the usual recursive object tracing scheme. This will be useful for comparison with our local tracing algorithm.

```
main()
   for each root r { trace(r) }

trace(p)
   o := object to which p points
   if isMarked(o), return.
   mark o
   for each valid pointer p' in o  { trace(p') }
```

The operation `mark` has its meaning specified by context. For example, a mark-and-sweep collector simply sets a bit corresponding to the object, while a copying collector moves this same object to a "live area." In any case, the only property we rely upon is that it is possible to test whether an object is marked using `isMarked`. We use this to ensure termination of the LTS process. It does not add to the principal idea of the optimization we propose. Instead, we focus on how objects are visited.

We observe that the algorithm presented above uses a depth-first traversal. While elegant, there are two problems with this technique:

The first problem is that recursion can be very deep and the associated overhead of stack activity can be expensive (allocation/deallocation of stack frames, context saving, *etc.*) This can be addressed with a combination of tail recursion, explicit stack management and pointer reversal. Pointer reversal temporarily modifies the heap, however, which creates problems in multi-threaded environments.

The second problem is that the topology of the graph of objects has a direct influence on traffic within the memory hierarchy. A traditional tracing algorithm does not take advantage of the relative locations of objects in the heap, possibly resulting in very bad locality of reference. For example, a page may be brought from disk to main memory to visit only one object even if other live objects are made accessible.

In this paper we demonstrate the possibility of improving on *both* aspects by transforming the depth-first tracing process into a "semi-breadth-first" one.

2.2 A Family of Tracing Algorithms

The principal idea behind our tracing technique is to defer visiting objects that lie outside a working set by maintaining queues of deferred pointers in fast memory (cache for example). When a queue becomes full, the deferred visits are made, altering the working set in a controlled fashion. This idea to localize the tracing process can be applied with minimal, localized modification to existing trace-based garbage collectors.

The deferred trace queues can be managed in a number of ways:

- One may keep all deferred object pointers in a common list, allowing or disallowing duplicates. When the list becomes full, it is analyzed to determine how to alter the working set. This has the advantage that the memory of the global queue is fully used, but the cost of the analysis may outweigh the benefit of making the optimal choice of working set alteration.
- One may associate a sub-queue to each range of addresses (heap region), with the number of ranges and size of sub-queues being parameters. Deferred object pointers are added to the appropriate sub-queue, either allowing or disallowing duplicates. When a queue is full, the associated region is added to the working set and visits are made. This has the advantage that deferring visits is fast, but the disadvantage is the deferred trace queue as a whole may be largely empty. This may be addressed by dynamically adjusting the size of the sub-queues based on use.

We have identified six strategies: { common list, static sub-queues, dynamic sub-queues} × { duplicate pointers allowed, not allowed }. We would expect the sub-queue strategies to be best when the far memory (RAM or secondary storage) speed is within a few orders of magnitude of that of the close memory (cache or RAM). Beyond this, we would expect the common list strategy to yield better results because here it is more important to avoid remote memory references.

Note that performing the deferred visits to one region may cause the trace queue of a second region to fill. At this point, starting to trace in the second region may cause the queue for the first region to fill. If both regions' deferred trace queues are nearly full and there are too many mutually referencing pages, local memory access can be lost. This situation degenerates to the usual handling of tracing, but with additional overhead. The problem may be avoided by taking one additional action: before performing the deferred marks on a region, the trace queue could be flushed to local store in the region itself or in a shared pool. This saved queue could then be substantially larger than the per-region queue maintained in fast memory.

2.3 Algorithm

We present a tracing algorithm where trace queues are associated with each heap region. This is the static sub-queues allowing duplicates strategy, described above. To allow fast access to these queues, they are contained in one contiguous area that we choose to be small enough to be maintained in fast memory.

Each region contains objects that will be marked and scanned. The difference from a regular tracing process is that scanning an object can reveal pointers *inside* the region currently collected or *outside*. If the pointer is to an object in the region, the object is visited recursively. When it points to another region of the heap, it means that following this pointer would not be optimal for the working set (or cache) behavior. In this case, we simply place this pointer in a trace queue for later examination. We thus maintain the working set for as long as possible, and reduce the number of cache misses or page faults.

When the process for a region is completed, we proceed to another region. The policy to determine the order in which regions are visited is implementation- or even application-dependent. It is likely, however, that choosing a region with a full or close to full trace queue will improve performance. A simple solution, avoiding the complexity of choosing the most populated queue, is to use a round-robin mechanism, and visit regions one by one. This is what we describe here.

In the initial step of the algorithm, roots are entirely dispatched into the different trace queues as if those pointers originated from an "external" region. Once the roots have all been recorded, the actual tracing begins. The complete algorithm is as follows:

```
mainTrace()
  initialRootsScan() -- initialize the trace queues
  while not all queues are empty
     { Q := choose a non-empty trace queue ; emptyQueue Q }

initialRootsScan()
  for each root r
     { Q := get trace queue for region where r points; enqueue(Q,r) }

emptyQueue(Q)
  while Q is not empty { p := dequeue Q ; followRef p }

followRef(p)
  o := object pointed to by p
  if (not isMarked(o)) { mark o ; trace o }

trace(o)
  for each valid pointer p in o
     if (p points to the same region as o)
        followRef(p)
     else
        { Q := get trace queue for region where p points; enqueue(Q,p) }
```

In the above, `enqueue` and `dequeue` are operations that add and remove elements from the trace queues.

2.4 Algorithm with Finite-Size Queues

We describe the static sub-queues strategy. In this scenario, it is required that a limit is placed on the size of the queues. We thus need to handle the problem of untimely full queues. In particular, when we visit a region and need to enqueue a pointer into a full queue, something must be discarded from the current working set to make room to work with the region with the full queue. Several strategies can be adopted:

- Empty the queue and deal with the pointer.
- Deal with the pointer first and then empty the queue.
- Empty a percentage of the queue and insert the pointer in the queue.
- Dump the queue to a reserved part of the region.

The *first* strategy is likely to be the safest, because the first action is to remove a pointer from the queue so it is not full anymore, thus allowing a new pointer to be enqueued. A situation where we need to add a new pointer to this queue can occur if, for example, the first visited object holds a pointer to a region which also has a full queue. In this case, this region is chosen to be visited and the first pointer may be to an object holding a pointer to the region we just visited. The *second* strategy allows following the new pointer first, thus removing the need to keep its information on the stack, but it is likely to become too costly in the case described above. The *third* strategy may be chosen when the working set is not entirely filled by pages of the current region. In this case, a certain number of pages can be brought into memory without dismantling the current working set. The *last* strategy may work in practice. In principle, however, the same problem must be considered in case the dump area overflows.

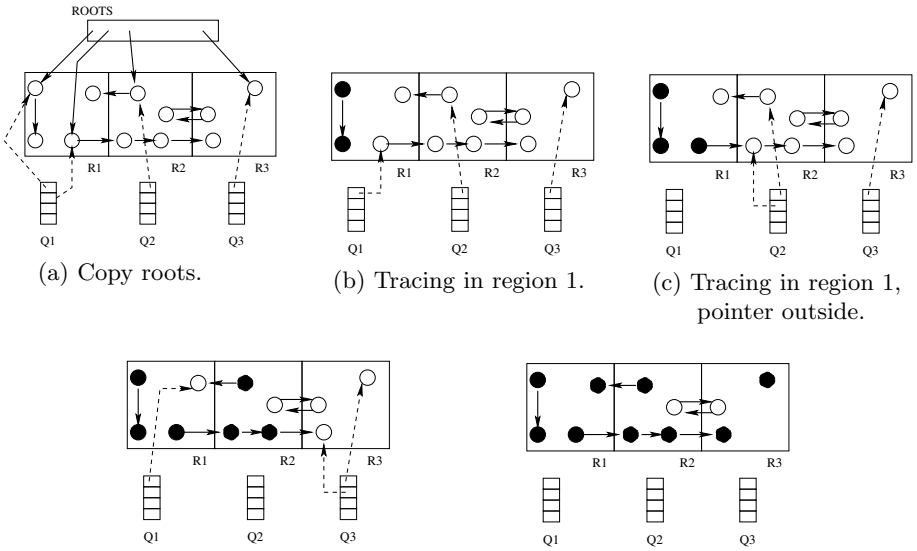
For simplicity, we choose to empty the queue first and then deal with the new pointer. The resulting algorithm is the same as that for unbounded queues, shown in Section 2.3, but with the calls to `enqueue` replaced by calls to `enqueueRef`, defined as follows

```
enqueueRef(Q, p)
  if (not full(Q)) enqueue(Q,p)
  else { emptyQueue(Q) ; followRef p }
```

3 Example

This section presents an example of the behavior of our algorithm. We follow the LTS process step by step.

1. First, the roots are copied into the trace queues. See Figure 1(a). (In GC, these are typically taken from registers, stack and intial static data area.)
2. Once the initial phase is completed, we see that two pointers have been recorded in the trace queue Q_1. We dequeue the first pointer and mark (using black coloring) the corresponding object. This object holds a pointer to another object in the same region R_1. We continue tracing along this path to mark the other object. See Figure 1(b).
3. We now use the second pointer recorded in Q_1. The object it points to is marked and scanned, and is found to hold a pointer to an object in R_2. This pointer is recorded in Q_2, as shown in Figure 1(c). Once this is done, we see that Q_1 is empty for now, so we continue the process with Q_2.
4. We retrieve each pointer of Q_2 and mark the objects, as we did for Q_1. We see in Figure 1(d) that Q_1 has been updated because an object in R_2 was pointing to R_1. Similarly, a pointer is also added to Q_3. Once Q_2 is empty, we visit Q_3.
5. Q_3 is visited and all reachable objects are marked. It is now empty, and we continue with Q_1. Once Q_1 has been visited, all queues are empty, and the tracing process is thus over, as shown in Figure 1(e).



(a) Copy roots. (b) Tracing in region 1. (c) Tracing in region 1, pointer outside.

(d) Tracing in region 2, adding to Q1 & Q3. (e) Finished all regions.

Fig. 1. Local Tracing Scheme example

We see that these trace queues act in a manner similar to entry items in a distributed garbage collection environment. Each pointer included in the queue indicates that an object of the region is reachable. All objects identified as live in a given phase of the LTS (depending on the graph of objects, there may be several phases) will be visited before starting the visit of another region, thus improving locality of treatment.

Another comparison can be made: our trace queues are simply remembered sets that are used to keep track of cross-boundary pointers. Note that trace queues may contain pointers to objects either marked (black, in the usual terminology), being marked (gray) or not yet traced (white), exactly like remembered sets.

4 Correctness of LTS

We provide an informal proof that LTS algorithm rephrases a regular GC mark phase algorithm. We first show correctness with the assumption that trace queues have infinite size (*i.e.* we can never reach a state where a queue is full). Later we treat the case of finite queues.

The LTS algorithm has two phases: initial root scan and trace phase. The code for the initial root scan is a simple recording of the roots. We show the trace phase is correct: that it is *safe* (marks all live objects), *complete* (does not mark any garbage) and *terminates*.

Termination: The number of times `trace` is called is bounded by the number of calls to `mark`, which is in turn bounded by the number of objects. After the initial root scan, `enqueue` is called only from `trace`, so the number of calls to `enqueue` is also bounded. The depth of recursion (through `followRef`) for a call to `trace` is limited by the bound on the number of calls to `trace`. Therefore any call to `trace` terminates, and so does any call to `followRef`. This, together with the fact that the number of calls to `enqueue` is bounded, gives termination of `emptyQueue` and `mainTrace`.

Safety: Because we have termination, we know that every pointer that is enqueued is eventually dequeued and handled. Handling a pointer to an unmarked object entails marking the object and enqueuing all the valid pointers it contains. All reachable objects are therefore handled.

Completeness: Only reachable objects are marked.

For the algorithm with bounded queues, we have:

Termination: The only obstacle to termination would be to indefinitely cycle through queues (emptying Q_a fills Q_b, so we need to empty Q_b, but this re-fills Q_a). We guarantee progression by dequeuing *first* (thus changing the state of the queue to "non-full"). Emptying a queue will treat at least one object, and since there are a finite number of objects it is impossible to indefinitely cycle through queues.

Safety: There is no loss of reference. The only case where this might happen with fixed size queues is when a queue is full and the reference we want to add to the queue is lost. However, the algorithm specifies that once the full queue has been emptied, we actually deal with this reference.

Completeness: The size of queues has no impact on the visited objects. We still start from the roots, in the same way the non-LTS algorithm does. Garbage is still guaranteed to be found.

5 Experiments and Results

5.1 The Test Environment

We implemented and tested LTS using the garbage collector of the Aldor language environment as a test harness. To support multi-language programming, the Aldor implementation employs a conservative mark-and-sweep GC. We compared the performance of various LTS-based mark phases with that of the usual depth-first mark phase. Our experiments have focused on improving paging performance, but we also made preliminary tests with cache-conscious configurations.

Finding appropriate standard benchmarks for garbage collection algorithms is quite difficult. We found that GC benchmarks are quite rare and macro-benchmarks usually focus on applications with modest memory footprint, e.g. 20-30MB (see [6] and [7]). In particular, we did not find any standard benchmark

using heaps larger than the normal size of physical memory. In order to permit careful study of LTS, we therefore constructed a set of micro-benchmarks. Each of these tested a particular kind of memory use. This allows greater understanding of the range of possible behaviours of LTS than macro-benchmarks would provide. In this context, we built a test suite that uses small programs by today's standards of desktop machines but that helps us confirm that the LTS is indeed an appropriate solution for large applications. Note that our tests are obviously not designed to represent real-life programs; rather, we have tuned them to exercise specific situations to help us better understand the limits of the LTS.

The precise nature of any improvement from LTS will of course depend on the relative speeds and sizes of the relevant two levels of the memory hierarchy. We expect the qualitative aspects to remain the same, however. Our tests were conducted with a 500 MHz Pentium III, with a UDMA66 hard disk and running Redhat Linux 7.1 (kernel 2.4.2). We were interested in testing our algorithm in an environment with a heap larger than physical memory. Since testing very large programs is time-consuming, we simulated the situation by working with programs using heaps of up to 178MB while limiting the amount of main memory available to the operating system to 32 MB. A preliminary form of these results was presented in [8].

5.2 The Benchmarks

Test 1: Fit in RAM (6MB used/32MB primary memory) The graph of objects fits entirely in RAM. There is no possibility of swapping so we expect no gain from LTS. This test allows us to quantify the overhead due to the extra management of regions.

Test 2: Linear structure (90MB used/32MB primary memory) Memory is filled with a linked list of large objects, with the links of the list in ascending address order. Here we observe paging, but the LTS does not change the order in which objects are visited. This test thus also allows us to quantify the LTS overhead.

Test 3: Parallel list creation (90MB used/32MB primary memory) Memory is filled with multiple parallel linked lists. Each list spans several consecutive regions and has its links in reverse memory order. All lists are pointed to by arrays in the first region. Depth first tracing would access objects in multiple regions for the first list, then the same regions again for the second list, *etc.* LTS should avoid this.

Test 4: Parallel list creation and use (178MB used/32MB primary memory) This test represents a more general memory situation. As before, lists are pointed to by arrays in the first region. Once all lists are created, a mutator loop is started. First, lists are swapped to allow the order of marking to be different from the original structure. Then, pointers to some lists are dropped, creating garbage. Finally, new parallel lists are created to re-populate the arrays. This is closer to a "real" application.

Test 5: Pointers everywhere! (178MB used/32MB primary memory) Here the arrays pointing to the lists are spread throughout the heap rather than being restricted to the first region.

Test 6: Cons-reversed lists (178MB used/32MB primary memory) Test 4 is modified to create lists linked in ascending memory order.

Test 7: Mixed-order lists (178MB used/32MB primary memory) Similar to Test 6, but only every second list is in ascending memory order. The others are in reverse memory order. The arrays pointing to the lists are still located at the beginning of the heap. This mixed order shows the behavior of the LTS in the presence of data structures that are accessible from different regions.

Test 8: Mixed-order lists with pointers everywhere (178MB used/32MB primary memory) This test is a combination of Test 5 and Test 7. Arrays are spread all over the heap while some lists are in reversed order and others are not. We hope to observe the behavior of the algorithm in presence of a graph of objects evolving in a less obvious manner than previous tests.

The details of the object sizes is as follows: Test 1 (Fit in RAM) used 600 linked lists, each of length 100. Test 2 (Linear structure) used 50 lists, each of length 15,000. All other tests used 3000 lists, each of length 500. The leaf objects contained in the lists were of three sizes: 16, 52 or 100 bytes.

5.3 Test Results

The timing results for the tests are displayed in Figure 2. For each set of parameters, the ratio of LTS to Non-LTS times is given. Tests were run three times each and the numbers shown here are the averages. We observed very little variation in the results (as can be expected because these tests are not random).

The table displays the test number as the header of each column. The row labels have the following meaning:

- *Non-LTS* corresponds to the results obtained with a regular tracing algorithm.
- *LTS-XX-YY* shows the results using the LTS with a region size of XX MB and a total size for the trace queues of YY KB. For example, LTS-4-512 corresponds to a test with a region size of 4MB and trace queues of 512KB.
- *Total app time* is the total application time, including the time taken by the Non-LTS GC.

We make the following observations:

Test 1: Fit in RAM This test illustrates a situation where there is no benefit in enhancing locality of reference. We measure that the overhead of maintaining the queues is about 25%. However, we note that this is for a total application time of 1 second, and the actual overhead is low in absolute terms. Also note that this overhead disappears if the GC is configured to use the LTS only when it can be beneficial.

	Test 1	Test 2	Test 3	Test 4
Non-LTS Marking	0.297	53	274	1222
LTS-4-256	0.366 (*1.23*)	62 (*1.17*)	127 (*0.46*)	322 (*0.26*)
LTS-4-512	0.366 (*1.23*)	65 (*1.23*)	128 (*0.47*)	317 (*0.26*)
LTS-4-1024	0.366 (*1.23*)	61 (*1.15*)	131 (*0.48*)	312 (*0.25*)
LTS-8-256	0.371 (*1.25*)	68 (*1.28*)	131 (*0.48*)	326 (*0.27*)
LTS-8-512	0.371 (*1.25*)	72 (*1.36*)	145 (*0.53*)	352 (*0.29*)
LTS-8-1024	0.372 (*1.25*)	69 (*1.30*)	141 (*0.51*)	344 (*0.28*)
LTS-16-256	0.371 (*1.25*)	83 (*1.57*)	139 (*0.51*)	338 (*0.28*)
LTS-16-512	0.371 (*1.25*)	75 (*1.41*)	141 (*0.51*)	350 (*0.29*)
LTS-16-1024	0.371 (*1.25*)	80 (*1.51*)	141 (*0.51*)	355 (*0.29*)
Total app time	1.000	108	404	1923

	Test 5	Test 6	Test 7	Test 8
Non-LTS Marking	1831	1217	1263	1869
LTS-4-256	755 (*0.41*)	341 (*0.28*)	352 (*0.28*)	1296 (*0.69*)
LTS-4-512	958 (*0.52*)	342 (*0.28*)	336 (*0.27*)	1311 (*0.70*)
LTS-4-1024	1061 (*0.58*)	347 (*0.28*)	350 (*0.28*)	1426 (*0.76*)
LTS-8-256	1145 (*0.62*)	363 (*0.30*)	329 (*0.26*)	1321 (*0.71*)
LTS-8-512	1025 (*0.56*)	358 (*0.29*)	332 (*0.26*)	1385 (*0.74*)
LTS-8-1024	1039 (*0.57*)	361 (*0.30*)	322 (*0.25*)	1434 (*0.77*)
LTS-16-256	872 (*0.48*)	368 (*0.30*)	338 (*0.27*)	1163 (*0.62*)
LTS-16-512	921 (*0.50*)	381 (*0.31*)	353 (*0.28*)	1144 (*0.61*)
LTS-16-1024	994 (*0.54*)	376 (*0.31*)	360 (*0.28*)	1156 (*0.62*)
Total app time	3038	1697	1946	3051

Fig. 2. Marking times (in seconds) for different parameters.
The LTS/Non-LTS ratio is shown in parentheses.

Test 2: Linear structure This test measures the other case for which the LTS approach is not well suited: With a single linked list, there are very few cross-region pointers and we see the basic LTS overhead. We see that by choosing carefully the size of the region and the queues, this overhead can be brought down to a reasonable level (15%).

Test 3: Parallel list creation This is the first test where we can observe the advantage of using the LTS. As explained before, lists in this example are created in parallel, resulting in many cross-region pointers. Following one list across several pages, then re-visiting the same pages for a second list, *etc.*, is very inefficient. LTS cuts the marking time by half and the total application time by a third.

Tests 4, 6, and 7: These tests give significant results: the structure of the lists in memory (from beginning to end, or end to beginning, or mixed) does not seem to influence the behavior of the tracing process. Here we see up to 75% improvement in marking time and a 50% improvement in total time.

Tests 5 and 8: Although speedups are less spectacular, they are still quite interesting: between 38% and 59% for *Test 5* and between 23% and 39% for *Test 8*. These results can be explained by the fact that "roots" (i.e the arrays that hold the lists) are scattered in memory. Instead of gathering all of them in the same set of pages, the GC has to swap extra pages in to reach these special objects.

5.4 Discussion

We observe a loss of performance for applications smaller than main memory or in which there are only a few objects with cross-region pointers. Although the overhead can be up to about half, it mostly concerns small applications which tend to be very fast anyway. For larger problems, however, speedups can be substantial, up to 75%.

Our algorithm performs better when swap space is involved. Small programs that fit in RAM do not need LTS. In fact, as emphasized by our experiments, our modifications will generate some overhead due to unnecessary actions such as tests to figure out if two objects are in the same region.

We propose a solution to avoid this overhead: we add a test before starting the tracing process. If the size of the heap is smaller than main memory, then LTS is not used. If the heap is larger than main memory, we activate the LTS. Alternatively, if paging statistics are available they may be used to trigger LTS.

At another level, it is also possible to activate LTS in a cache-oriented configuration when the heap is smaller than main memory. Further experiments are required to understand the merits of such an approach.

There are two parameters that can be tuned to control the behaviour of the LTS: the size of regions and the size of trace queues. The main issue is the choice of the optimal size of "window of collection" (or "region"). A region should be large enough to avoid the need for large trace queues and small enough both to avoid thrashing and to keep a reasonable working set. Obviously, there is no one best choice, as the size of a region largely depends on the nature of the applications. In our experiments, we found that region sizes of 4MB gave the best results most of the time, but this is not always the case (see for example *Test 7* and *8*). The second parameter is the size of the trace queues and this is dependent on the size of a region.

6 Multiprocessor LTS

The LTS organizes the heap in such a manner that parallelization becomes natural. The heap is divided into regions that can each be mapped to a thread or a processor. In this section, we discuss various aspects of using the LTS in a multiprocessor environment.

It is straightforward to assign regions to be handled independently and in parallel by threads on separate processors. Each thread can scan a group of regions repeatedly and update the different trace queues. Although performance is likely to improve due to the parallel nature of processing, the organization of the

memory hierarchy can be more complex in a multiprocessor, so specific working set considerations will be architecture-dependent. The single processor LTS optimization controls the working set to reduce inefficiencies within the memory hierarchy during tracing. A shared memory multiprocessor version should strive to preserve this essential characteristic.

When the LTS is configured to improve cache behavior, its multiprocessor performance should also be improved. This results from a useful property of certain multiprocessor environments: While the heap may be common to all processing units, there is usually at least one level of per-processor cache. When several processors are used, each of them will use its cache while accessing objects. An advantage of the LTS is that cache consistency is maintained very simply by the assignment of a range of regions to each processor. A given processor will never visit an object in a region assigned to another one (except in the case of work stealing as described below, but in this case the region can be reassigned to another processor).

The only synchronization required is to manage accesses to trace queues and to identify termination. A simple idea to discover termination is to maintain a counter of threads going to sleep when no more work is available. If a thread adds a pointer to a queue, it wakes up the thread associated to that region. Termination occurs when the last thread goes to sleep. If a thread appears to be the last one going to sleep, it synchronously checks the counter and the queues to make sure no reference has been left behind.

A final issue is that of load balancing among processors. It is likely that regions will be unequally populated. One region may hold a large number of objects, while others contain no or few objects. In this case, some processors will starve due to the lack of work. Endo [9] proposed a solution in the form of "work stealing"[10]. In this case, each thread maintains a "work queue" containing pointers that the thread should examine next. Once it is empty, there are two possibilities: Either the thread goes to sleep until something has been put in its queue or the thread helps other threads by "stealing" pointers from their queues and inserting them into its own queue.

If work-stealing is used naïvely, the parallel version of the LTS reverts to Endo's technique where several processors scan a single region, involving a synchronization mechanism to access objects. This can be avoided by making regions small enough to assign several regions to one processor. In this case, regions – instead of pointers – can be stolen. This requires a simple locking mechanism at the level of regions rather than objects. We believe this coarser-grained approach could lead to a significant improvement over Endo's results in some cases.

7 Related Work

This section presents garbage collection techniques – both in uniprocessor and multiprocessor contexts – that we can relate to the LTS.

Although it is more generally applicable, we have presented LTS primarily in the context of mark-and-sweep (M&S) collectors. Some have argued that with recent advances in GC technology, M&S is no longer important. We feel otherwise. Boehm [11] and Zorn [12] argue that stop-and-copy collectors do not necessarily perform better than M&S. Particularly, Zorn compares both techniques in a generational setting and concludes that M&S typically uses 20% less memory than stop-and-copy, but was only 3%-6% slower on the problems he tested. While a copying collector apparently improves locality over time, these analyses show that this factor is not sufficient to clearly improve performance. From another perspective, it is increasingly common to use hybrid techniques to combine the attractive features of various methods; M&S collectors figure prominently in this setting. Finally, in some settings, *e.g.* with heavy use of non-GC aware foreign libraries, conservative M&S is the only viable option.

Generational algorithms divide the heap into "regions" (called generations) to reduce to a minimum the work done by the collector at each call. Because each collection of the nursery is focused in a small area of memory, a side-effect of this organization is to localize data treatment thus reducing page faults and possibly cache misses. Collecting the old generation often involves collecting the entire heap. This is sometimes done with M&S and sometimes with other techniques. In either case, the LTS can be used in the same way as with non-generational algorithms. We would then benefit from the use of generations *and* of an improved trace process for the collection of old generations when large heaps are collected.

The observation that collecting the old generation is disruptive has been previously made in MOS [13]. This incremental GC precisely defines the memory block to examine at each call of the collector for the old generation. It is claimed that this allows a more suitable solution for real-time applications, for example. While the LTS does not solve the problem of real-time applications, we believe it proposes a simple, useful technique to reduce the time spent in collecting the old generation.

Attardi's CMM [14] proposes a heap organization similar to the LTS but for a different purpose. In CMM, each region of the heap is associated with a specific memory management scheme. This allows potential use of a different GC for each sub-heap. Consequences for paging and caching behaviors were not considered. The point of view proposed by the LTS could be used to CMM's advantage. The natural technique used by CMM is to allow collectors to follow pointers even in other sub-heaps to possibly discover live objects in the current sub-heap. Such out-of-sub-heap pointers could be buffered in trace queues to preserve the working set of the collector, which is the job of the LTS.

In [15], Boehm studies a technique to improve caching behavior during tracing of a mark-and-sweep garbage collection. It relies on a standard hardware feature (which can be found on Intel and AMD platforms, as well as HP RISC machines)

to pre-fetch "child" objects into the cache when an object is examined. When the object is required by the tracing process, it is already in cache. In comparison, the LTS improves another aspect of tracing. Instead of importing objects before they are needed, it keeps objects in cache as much as possible to increase the probability they will be available in case they are needed. It is likely that both techniques could be combined.

Boehm [15] also mentions an improvement of the sweep phase that uses a bitmap to mark dirty pages (*i.e.* containing live objects). When sweeping memory, the GC checks the bitmap before examining a page in detail to rebuild its free list of fixed-sized objects. If the bit is not set, the page can be reclaimed as a whole. The LTS provides a simple solution to store the bitmap: it may be placed in the trace queues. In addition to storing pointers, we maintain a bitmap of pages in the same memory area. This is useful because trace queues are designed to fit in main memory (or cache), which also allows fast access to the bitmap. The overhead is of 1 bit per page, that is 512 bytes for a region of 16MB. Preliminary experiments showed up to 55% improvement with the Aldor compiler.

The idea of optimizing paging access during garbage collection was mentioned in [16]. The original objective was to improve performance when collecting the old generation in a generational garbage collector. The principle was to partition the heap into sections called buckets that are similar to regions in the LTS. As mentioned in the paper, limiting the trace activity to one card at a time is also a solution to avoid paging. This may have the drawback of maintaining bookkeeping information (incoming out-of-card pointers) for a possibly long time.

Hertz *et al* [17] propose an alternative solution to control paging access. Their garbage collector is a generational collector using "book-marking" to keep pages in main memory as much as possible. The proposed mechanism is quite precise as it associates actions to swapped-out pages. This technique requires modifications of low-level layers to gain control over the paging system. The LTS takes a different approach. Although less precise, the solution proposed by the LTS is simple to put in place and does not require low-level modification of a virtual machine or operating system: The idea behind LTS is to keep working with the same set of objects as long as possible. This means corresponding pages will stay in faster memory for long periods. The LTS approach does not rely on interfaces that (if available) will differ by operating system and memory hierarchy level.

Multiprocessor parallel collectors do not benefit from the same attention as concurrent GCs. However, several techniques were studied: [9] and [18], for example. An advantage offered by the LTS compared to the parallel collector described by Endo *et al* in [9] is that there is no need for synchronization at the object level. Even though Endo proposed an optimization to access these objects, a synchronization mechanism is still required. This can lead to a costly marking process (although this aspect is not the only issue, as observed in the paper). Instead of asking each processor to trace a given data structure from

beginning to the end, the LTS limits the activity of each processor to regions of memory. If a structure steps over a frontier, the rest of its tracing is handled by another processor. This removes the need for complex synchronization at this level.

We also note that, as mentioned in Section 6, the LTS offers a simple organization of the heap suitable for a parallel configuration. The advantage is that uniprocessor and multiprocessor environments requiring mark-and-sweep could use the same memory management technique with very little modification, *and* receive interesting performance optimization.

8 Conclusions and Future Directions

In this paper we described what we have called a "Localized Tracing Scheme," a technique to improve performance of tracing activities such as those used in garbage collectors. The LTS localizes the tracing process by dividing memory into regions and deferring out-of-region tracing. The idea of deferred pointer queues is simple to implement and can be readily added to existing collectors.

LTS limits the working set to a region of the heap rather than the entire heap. If a region can fit largely or entirely in cache, cache misses are reduced. In the same way, if the region is smaller than available RAM, thrashing due to page faults diminishes. Consequently, optimizations can be made at different levels of the memory hierarchy: cache, virtual memory and network.

We have tested this strategy in the context of the Aldor garbage collector, using a suite of specific micro-benchmarks to observe the behaviour of the LTS in practice. We obtained up to 75% improvement with a configuration oriented towards virtual memory optimization.

Finally, we presented how LTS can function in a multiprocessor context. We observed two axes: (i) independently of any optimization, the organization of the heap in regions results in a natural setting for parallel garbage collections, and (ii) parallel GCs in multiprocessor environments may be improved at cache and virtual memory levels.

We are interested in a number of directions suggested by LTS. First, it has become a practice in scientific computing to deduce and optimize hardware-related parameters through dynamic tuning of algorithms. This may be a useful approach to determine the best sizes for memory regions and deferred pointer queues. Second, GC implementations often maintain an explicit tracing stack, rather than relying on functional recursion. It would be interesting to study different ways of combining deferred pointer queues and the explicit tracing stack. Third, it would be useful to better understand the performance trade-offs arising in different strategies to flush full queues. Fourth, with multi-core processors becoming the norm for personal computing, a full implementation of the multiprocessor LTS would be of practical interest. Finally, the LTS was developed for the Aldor language for computations in computer algebra, which are typically very demanding in dynamic memory use. It would be useful to implement LTS in a more mainstream environment, such as the MMTk [19], allowing direct comparisons over a wider range of common benchmarks and experiments in conjunction with other memory management strategies.

References

1. Watt, S.M.: Aldor. In Grabmeier, J., Kaltofen, E., Weispfenning, V., eds.: Handbook of Computer Algebra, Springer Verlag (2003) 265–270
2. Aldor.org: Aldor user guide. `http://www.aldor.org/AldorUserGuide` (2003)
3. Maplesoft: Maple User Manual, Maplesoft, a division of Waterloo Maple Inc. (2005)
4. Boehm, H.J., Weiser, M.: Garbage collection in an uncooperative environment. Software Practice and Experience **18** (1988) 807–820
5. Lieberman, H., Hewitt, C.E.: A real-time garbage collector based on the lifetimes of objects. Comm. ACM **26(6)** (1983) 419–429. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980.
6. Boehm, H.J.: GCBench. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench`
7. Grunwald, D., Zorn, B.: Malloc benchmarks. `ftp://ftp.cs.colorado.edu/pub/cs/misc/MallocStudy`
8. Chicha, Y.: Practical Aspects of Interacting Garbage Collectors. Ph.D. Thesis, University of Western Ontario (2002)
9. Endo, T., Taura, K., Yonezawa, A.: A scalable mark-sweep garbage collector on large-scale shared-memory machines. In: Proc. High Performance Computing and Networking (SC'97). (1997)
10. Burton, F.W., Sleep, M.R.: Executing functional programs on a virtual tree of processors. In: Proc. the 1981 Conference on Functional Programming Languages and Computer Architecture. (1981) 187–194
11. Boehm, H.J.: Mark-and-sweep vs. copying collection and asymptotic complexity. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/complexity.html`
12. Zorn, B.: Comparing mark-and-sweep and stop-and-copy garbage collection. In: Proc. 1990 ACM Symposium on Lisp and Functional Programming. (1990)
13. Hudson, R.L., Moss, J.E.B.: Incremental garbage collection for mature objects. In: IWMM'92 Proceedings. (1992)
14. Attardi, G., Flagella, T.: A customisable memory management framework. Proc. USENIX C++ Conference, Cambridge, MA. (1994)
15. Boehm, H.J.: Reducing garbage collector cache misses. In: ISMM 2000 Proc. Second International Symposium on Memory Management. (2000)
16. Demers, A., Weiser, M., Hayes, B., Bobrow, D.G., Shenker, S.: Combining generational and conservative garbage collection: Framework and implementations. In: Proc. ACM Symposium on Principles of Programming Languages, San Francisco, California, ACM Press (1990) 261–269.
17. Hertz, M., Feng, Y., Berger, E.D.: Garbage collection without paging. In: Proc. SIGPLAN 2005 Conference on Programming Languages Design and Implementation, Chicago, IL, ACM Press (2005)
18. Taura, K., Yonezawa, A.: An effective garbage collection strategy for parallel programming languages on large scale distributed-memory machines. In: ACM Symposium on Principles and Practice of Parallel Programming. (1997) 264–275
19. Blackburn, S.M., Cheng, P., McKinley, K.S.: Oil and water? high performance garbage collection in Java with MMTk. In: ICSE 2004, 26th International Conference on Software Engineering, Edinburgh (2004)