Journal of Universal Computer Science, vol. 13, no. 6 (2007), 817-829
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

A New Architecture for Concurrent Lazy Cyclic Reference Counting on Multi-Processor Systems

Andrei de Araújo Formiga
(Universidade Federal de Pernambuco, Brazil
andrei.formiga@gmail.com)

Rafael Dueire Lins
(Universidade Federal de Pernambuco, Brazil
rdl@ufpe.br)

Abstract: Multi-processor systems have become the standard in current computer architectures. Software developers have the possibility to take advantage of the additional computing power available to concurrent programs. This paper presents a way to automatically use additional processors, by performing memory management concurrently. A new architecture with little explicit synchronization for concurrent lazy cyclic reference counting is described. This architecture was implemented and preliminary performance tests point at significant efficiency improvements over the sequential counterpart.

Keywords: memory management, garbage collection, reference counting, concurrent garbage collection
Categories: D.1.3, D.3.4

1 Introduction

Automatic memory management (often called *garbage collection*) has become widespread in current programming systems as witnessed by the wide adoption of garbage-collected execution environments such as Sun Java and Microsoft .NET platform. Automatic memory management programming systems improve programmer productivity and the reliability of resulting programs by relieving programmers from the burdensome and error prone task of having to manage memory manually [Jones and Lins, 1996].

Most current systems, implemented on sequential architectures, use a sequential garbage collector, where memory management tasks alternate with tasks related to the user process; these two tasks are never executed concurrently. However, current trends in processor design and marketing indicate that single-chip multiprocessors are becoming of widespread use in recent computer systems. Such change in computer architecture opens the possibility of taking advantage of the extra computing power available. Executing memory management tasks concurrently with the user processes, programs can automatically take advantage of the available processors on a computer, with no changes required to the user programs.

Reference counting, first developed by G.E. Collins [Collins, 1960], is one of the main methods for automatic memory management. This paper proposes a new architecture for a concurrent reference counting memory management system, based

on a similar proposal by Lins [Lins, 2005]. The motivation for a new architecture was the need to minimize explicit synchronization between the collector and the rest of the program, particularly to avoid the use of locks; Lins' original proposal was the basis of the garbage collector of several commercial machines, but their code was never freely available. The architecture proposed herein was implemented in a real programming system, and preliminary performance tests show promising gains in overall performance of programs.

This paper is organized as follows: Section 2 briefly presents the reference counting algorithm in its simplest form and some milestones in its development, leading to previous architectures for parallel and concurrent reference counting, some of which were used as the basis for the one presented here; Section 3 details the new architecture for concurrent reference counting on multiprocessors that minimizes explicit synchronization, and emphasizes its differences in relation to its predecessors; the architecture in Section 3 is limited to one collector and one user process, thus Section 4 suggests ways in which the proposed architecture may be extended to work with multiple user processes, and multiple collector processes; finally, Section 5 presents the current implementation of the proposed architecture and the results of some initial performance measurements.

2 Reference Counting

The reference counting method consists of associating with each object in the heap a counter that stores the number of references to it. An object is active when it is currently in use and somehow accessible by the user program, which is indicated by it being transitively connected by references to one of the roots of the computation. The user process alters the connectivity of the objects both increasing and decreasing their reference count. Whenever the counter of an object reaches zero, it means that it is no longer active thus it is *garbage* and may be reclaimed to be later reused. It is usual to consider the heap as organized in a directed graph, where objects are the nodes and references are the edges. After Dijkstra [Dijkstra *et al.*, 1976] it is common to call *mutator* the part of the program that is concerned with the user process and *collector* the garbage collection system.

A distinctive advantage of reference counting over other methods of garbage collection is that its operations are interleaved with mutator activity, resulting in a technique that is naturally incremental. Other techniques, like mark-scan [McCarthy, 1960] and copying collection [Fenichel and Yochelson, 1969], require that the mutator stops for some time while the collector performs its work; this leads to pauses in interactive systems, which may be a nuisance. However, standard reference counting has a serious drawback: it can not reclaim cyclic data structures, as noticed by J.H.McBeth in 1963 [McBeth, 1963]. Thereafter, many solutions to this problem were proposed, but either they were not general-use, or were actually a proposal to use two memory management systems, one of them solely to reclaim cycles. The first general solution to the problem of reference counting cyclic structures was published in 1990 by Martinez, Wachenchauzer and Lins [Martinez et al., 1990]; the idea is to identify points in the graph where cycles can be formed, and perform from there a local mark-scan to detect potential space-leaks, garbage cycles. In subsequent papers, Lins extended this solution in many ways, for instance by making the local cycle

analysis lazy [Lins, 1992a] and by identifying critical points in the graph that could speed up the analysis [Lins, 2002].

To keep track of the state of local analysis in a local mark-scan, objects have to maintain not only a reference count, but also a color. The current color of a cell indicates its state regarding the local mark-scan.

Lins also developed a series of parallel reference counting algorithms targeted at multi-threaded or multi-processor systems [Lins, 1991; Lins, 1992b; Lins, 2005], but none of them were implemented in real multi-processor architectures. However, they were the basis for the two IBM Java machines implemented at IBM-T.J.Watson [Bacon et al., 2001] and IBM-Haifa [Levanoni and Petrank, 2006]. The architecture presented in this paper is based on the latest version of Lins' algorithm for parallel multi-processed lazy cyclic reference counting [Lins, 2005], detailed in the next section. This new architecture was implemented, with promising performance results, as detailed in Section 5 of this paper.

3 The Proposed Architecture

The new architecture proposed in this paper, in its basic version, is designed to use two processors executing concurrently: the mutator and the collector. Figure 1 shows a schematic view of this architecture. The two processes share access to three queues: a list of free objects, an increment queue and a decrement queue. The mutator alters graph connectivity getting memory from the *free-list* whenever a new object is created. When new references are made, the mutator inserts increment requests onto the *increment queue*; likewise, when references are destroyed, the mutator inserts decrement requests onto the *decrement queue*. The collector manages memory, removing and processing requests for increment and decrement from the appropriate queues and detecting when objects become garbage; when this happens, memory reclaimed from garbage objects is added to the free list. A detailed presentation of how the algorithm works is presented in the next subsections. For a matter of simplicity of the management of the free-list, it is considered that objects are fixed-size cells.

3.1 Mutator Operations

From the point of view of garbage collection, the mutator can only create new objects (getting cells from the free-list), create new references to objects, or destroy existing references to objects. Besides standalone operations for creation and destruction of references, it's often useful to have a single update operation that combines the deletion of a reference to an object and the creation of a reference to another object – e.g., when a pointer that references an object is changed to point to another one – a reference to the former object is destroyed, while the latter has a new reference to it. So, in this model the mutator performs three operations:

- `New` – to allocate new cells, getting them from the free list
- `Del(S)` – used when a reference to cell `S` is destroyed; generates a decrement request

- `Update(R, S)` – used to update a reference from object R to object S

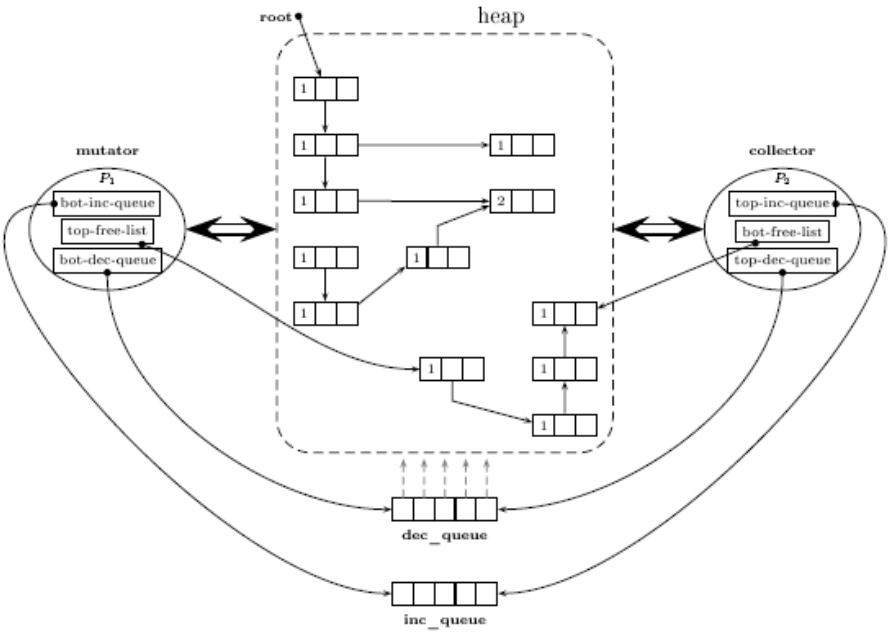

Figure 1: Basic architecture for concurrent reference counting with one mutator and one collector

The mutator never changes directly the reference count of a cell, nor its color. All memory management is done by the collector. This has implications for the necessary synchronization of mutator and collector, as will be seen later.

The algorithm for operation `New` is presented below. The mutator simply checks the existence of cells in the free list and gets one if available; otherwise it remains busy waiting for new cells. This is reasonable, because in this case the mutator can not continue its computation, and must wait for the collector to make cells available.

```
New() =
   if not empty(free-list) then
      newcell := get_from_free_list()
   else
      newcell := New()
   return newcell
```

The operation `Del` is even simpler than in standard reference counting: the mutator only inserts a new decrement request into the decrement queue. The collector takes care of the remaining graph adjustment.

```
Del(S) =
       add_decrement(S)
```

Finally, `Update` must replace a reference by another, thus it must call `Del` to notify the destruction of a reference, and insert a reference onto the increment queue.

```
Update(R, S) =
   Del(*R)
   add_increment(S)
   *R := S
```

It is easy to see in the operations above that the mutator only removes cells from the free list (actually organized as a queue) and inserts elements onto the increment and decrement queues. This results in simpler management of the three queues, shared by two concurrent processes.

3.2 Collector Operations

The memory management responsibilities, for a reference counting system, are to keep track of reference counts and detect when objects become garbage. It is also necessary to manage the cycle analysis, performed by the local mark-scan procedures. In the proposed architecture, the collector must process increment and decrement requests placed on the corresponding queues; adjust reference counts accordingly; detect any cells that have become garbage and insert them onto the free list; and besides that to keep track of possible cyclic structures that may have become garbage. This outline of how the collector works is reflected in its main operation, `Process_queues`, detailed below.

The analysis of cyclic structures requires that cells must have one of three possible colors: green, red and black. Green cells are active cells that are not scheduled for a local mark-scan. Red cells are currently under a local mark-scan. Finally, black cells are marked and scheduled for a local mark-scan at some point in the future. Only shared cells may be part of a cycle, and such a cycle becomes garbage when a shared cell has its reference count decremented. This means that when a cell has its reference counted decremented to a value greater than zero, it must be marked for future analysis. Cells that were marked for a local mark-scan are kept in a data structure called the *status analyzer*. It is a collection of cells that were marked black and must be analyzed later on. The original algorithm by Martinez, Wachenchauzer and Lins [Martinez et al., 1990] was eager: whenever a shared cell had its counter decremented, it was immediately analyzed with a local mark-scan. Experience has proved that doing the analysis lazily, as proposed by Lins [Lins, 1992a], yields far better performance.

When a cell in the status analyzer is selected, a local mark-scan takes place: the cell and its sub-graph is painted red and their reference counters are decremented, eliminating references that are internal to this sub-graph. If there remain cells with

reference counter greater than zero at the end of this red-marking phase, it means that there are external references to the sub-graph, thus it is necessary to paint cells green and restore their counters. If no red cell in the sub-graph has a counter greater than zero, the whole sub-graph is a garbage cycle and must be collected so that the cells can be returned to the free list. The status analyzer is also used to keep objects that are critical points in the object graph, points that when analyzed will allow the algorithm to decide in less steps whether a sub-graph is cyclic garbage or not. This is a very brief description of the cyclic reference counting algorithms developed by Lins. For further details one may refer to [Lins, 2002].

The main operation in execution by the collector is `Process_queues`, whose basic function is to process requests for increment and decrement of reference counts, getting them from the appropriate queues. Adjustment of reference counter in objects will then trigger most other actions of the collector, like freeing the memory occupied by garbage objects and marking objects for cycle analysis. The only operation that will not be triggered by adjusts in reference counters is scanning the status analyzer to detect cycles, so this is included in `Process_queues`: it first processes the increment queue, taking increment requests from it and performing them (cells are assumed to have a reference counter field `rc` and a color field `color`); then it processes the decrement queue, calling operation `Rec_del` to destroy a reference; finally, after processing both queues, the status analyzer is processed by a call to `scan_status_analyzer`. Operation `Rec_del` must be used to delete a reference because if the cell has its counter hit zero, all its outgoing references must be deleted recursively. The operation `Process_queues` is defined by the following pseudo-code listing:

```
Process_queues() =
   while not empty(inc-queue)
      S := get_increment()
      S.rc := S.rc + 1
      S.color := green
   if not empty(dec-queue) then
      S := get_decrement()
      Rec_del(S)
   else
      scan_status_analyzer()
   Process_queues()
```

The recursive reference deletion operation, `Rec_del`, is defined so that it first checks to see if the removed reference is the last one to the object; in this case, it must be deleted and its outgoing references removed recursively; the field `children` in a cell is supposed to be a collection of all its outgoing references. If the cell has a reference count with value greater than one, it is either a potential candidate to have an external reference transitively connecting it to root or being a knot-tying point to a cycle, and thus it must be painted black and added to the status analyzer; a later scan of the status analyser will reveal which one is the case. Thus, the complete definition of this function is:

```
Rec_del(S) =
   if S.rc == 1 then
      S.color := green
      for T in S.children do
         Rec_del(T)
      add_to_free_list(S)
   else
        S.rc := S.rc - 1
      if S.color != black then
         S.color := black
         add_to_status_analyzer(S)
```

The remaining operations of the collector are all related to the analysis and detection of cyclic garbage. `scan_status_analyzer` will be called whenever there are no increment or decrement requests to be processed (in the worst case, it will be called because of memory exhaustion – the mutator will be blocked if it can not allocate a new object). Its definition is presented below. When a cell S is in the status analyzer, it may be either black or red. If it is black, a local mark-scan is scheduled to be performed in the sub-graph below S. Operation `mark_red` is called to start the local analysis. If the cell is red, it means it was identified in `mark_red` as a critical point that may be connected to external references; its counter is checked, and if it is greater than zero the sub-graph below it is active, so it is necessary to undo the effects of operation `mark_red` by calling `scan_green`. The function calls itself recursively, looking for further items to process; if there are none left, the remaining cells still colored red are garbage in cyclic structures, so `collect` is called to recycle them.

```
scan_status_analyser () =
   S := select_from(status_analyser)
   if S == nil then return
   if S.color == black then
      mark_red(S)
   if S.color == red && S.rc > 0 then
      scan_green(S)
   scan_status_analyser ()
   if S.color == red then
      collect(S)
```

`mark_red` performs the red-marking phase of the local mark-scan. It tests if the cell is red, coloring it red if not; then the reference counter of all its children (objects referenced by it) is decremented, canceling references that may be internal to a cycle. `mark_red` is called recursively through the whole sub-graph of S. If a cell is already painted red – which means it already had its reference counter decremented – and still has a counter with value greater than zero, it is a critical point in the graph that may be connected to external references. Adding it to the status analyzer will speed up the process of deciding if a sub-graph is a garbage cycle or not [Lins, 2002]. This

accounts for red-colored cells found in the status analyzer, as seen previously while describing `scan_status_analyzer`.

```
mark_red(S) =
   if S.color != red then
      S.color := red
      for T in S.children do
         T.rc := T.rc - 1
      for T in S.children do
         if T.color != red then
            mark_red(T)
         if T.rc > 0 && T not in status-analyzer then
            add_to_status_analyser(T)
```

If a sub-graph is found to have external references, it is necessary to restore counters decremented by mark_red; This is the task of `scan_green`. It paints a cell green and increments the counter of its children.

```
scan_green(S) =
   S.color := green
   for T in S.children do
      T.rc := T.rc + 1
      if T.color != green then
         scan_green(T)
```

The only remaining operation to be described is `collect`, whose responsibility is to free the memory associated with each garbage object detected in a sub-graph. Its definition is shown below. Cells returned to the free list are painted green and assigned a reference counter with value 1, to simplify cell initialization when they need to be later reallocated.

```
collect(S) =
   for T in S.children do
      Delete(T)
      if T.color == red then
         collect(T)
   S.rc := 1
   S.color := green
   add_to_free_list(S)
```

3.3 Synchronization

Now that the operations of both processes were described, it is now considered how they can work concurrently in a safe and efficient manner. These operations were adapted from the sequential version of the cyclic reference counting algorithm [Lins, 2002], and the whole architecture presented here was based on that of Lins [Lins, 2005]. Lins' architecture used only two shared queues between collector and mutator,

the free list and a decrement queue (called a *delete queue* in the original paper). The result is that both collector and mutator read and update reference counts in cells, so this configures a potential interference between them [Andrews, 1999]. Lins dealt with this by assigning priorities to the processors, so that one would always be allowed to access reference counter fields before the other in a situation of contention. This would be a good solution, if current hardware provided a way to ensure the priorities. However, this is not the case in current shared-memory multi-processor architectures, so implementing Lins' architecture in current hardware would create the necessity of explicit synchronization between collector and mutator when reading or updating reference counts in cells, and as these are frequent operations, the overhead involved with synchronization would compromise performance. The architecture presented here solves this by using a technique of disjoint variables [Andrews, 1999], where only the collector reads and updates the values of reference counts in cells, and only the mutator changes the value of references in the memory graph. This works by having two queues, one for requests for increment and another for requests of decrements, shared between the two processes. The resulting architecture has little need for explicit synchronization. None is needed for reads and updates of reference counts or reads or updates of pointers.

A remaining concern regarding synchronization is about the three shared queues. Although it is not described here, there are known ways to implement concurrent queues with little need for explicit synchronization, especially for the case of a single reader and single writer [Valois, 1994]. The chosen implementation of concurrent queues will dictate how operations like `add_increment`, `add_decrement`, `get_from_free_list` and their counterparts will be ultimately implemented.

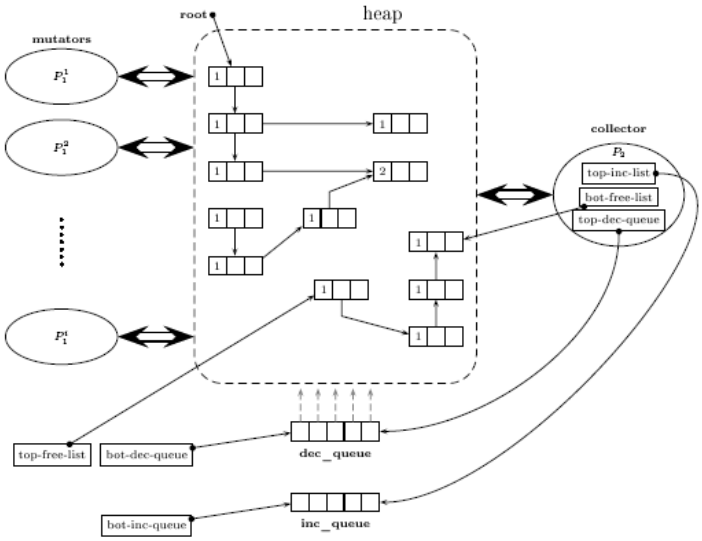

Figure 2: Architecture for concurrent reference counting with more than one mutator process

4 Multiple Mutators and Collectors

In Section 3 the basic architecture for concurrent reference counting was presented, considering a designation of one process as the mutator and one as the collector. In many programs, it may be desirable to have more than one mutator or thread executing concurrently, thus the architecture is now generalized. Figure 2 shows a schematic version of this multi-mutator architecture.

The main difference in relation to the single mutator version is that now all mutators must share access to one end of the three queues. To accomplish this, mutator processes no longer access the queues directly, but through shared registers, called `top-free-list`, `bot-dec-queue` and `bot-inc-queue`. Access to these registers must be synchronized to avoid interference; this can be done by using *compare-and-swap* type operations, present in hardware in most current processors. Implementation of lock-free queues with many readers (for the free list) or many writers (for the request queues) is still possible [Valois, 1994]. Otherwise, the algorithms work just like the single mutator case. In particular, the collector does not need any changes to accommodate this architecture.

Another direction of extension to the architecture presented in Section 3 is to allow for many collector processes to work cooperatively in a parallel garbage collection architecture. This could be combined with the suggestions for multiple mutators to obtain a multi-mutator, multi-collector architecture. The issues involved in such architecture, however, are more complicated and include greater problems of interference and load-balancing between collectors, and between them and the mutators. This possibility is not considered further in this paper. Previous work by Lins [Lins, 2005] presents such architecture in greater detail.

The multimedia retrieval is completely separated from the multimedia database, according to the objective of modularity. In our prototype a Web interface is used to specify queries and to display the search results. The Web server forwards the queries of each user to a broker module. This broker communicates with the multimedia database, groups received search results, and caches binary data. The usage of broker architecture makes it possible to integrate other (e.g. text) databases or search engines by simply adding the brokers of these data sources to the Web server. Vice versa also other search engines can have access to the multimedia database by using the broker from this system.

5 Implementation and Results

The architecture of Section 3 was implemented as the garbage collection sub-system for a programming language. A compiler for a lazy, purely functional programming language (similar to a subset of Haskell) was written to generate benchmarks for the implemented garbage collection system. The compiler uses well-known techniques for implementation of functional programming languages [Peyton-Jones, 1987] and generates code for an execution environment similar to the G-machine; this runtime environment includes an implementation of the garbage collector described in Section 3.

This implementation was used in experiments to assess the performance of the new architecture with relation to the sequential version of the same algorithm. Table 1 shows the results for execution of six test programs compiled to work with a sequential reference counting algorithm. All tests were executed in a single 1.6GHz Athlon MP computer with 2 processors and 512Mb of RAM, with a free list of 5000 cells and a status analyzer structure that could hold a hundred items.

Benchmark	alloc	scan_sa	mark_red	scan_green	collect	time(s)
acker	253175	4013	40574	40127	447	0,0351
conctwice	42834	521	5468	5406	62	0,005
fiblista	14837640	27892	857421	836317	21104	3,0812
recfat	335959	4985	49872	49338	534	0,0872
somamap	94309	1356	10521	10409	112	0,0473
somatorio	35298	499	4987	4933	54	0,0274

Table 1: Results for the execution of six test programs with sequential reference couting

The tests where selected for being highly recursive programs, and in some cases for using many list objects. Recursion was implemented by knot tying the Y-combinator, yielding cycles in the graph-reduction machine [Turner 79] which is part of the execution environment. Programs with many recursive calls generate many cycles, exercising the cycle analysis capabilities of the algorithm. The last column on Table 1 reports overall execution (CPU) times, in seconds. The other columns list the number of calls to operations related to memory management: the **alloc** column contains the number of allocated cells during the execution of the program; **scan_sa** is the number of calls to `scan_status_analyzer`; **mark_red**, **scan_green** and collect each record the number of times the respective operations were called.

The results on Table 1 make sense only when compared with those of Table 2, which shows results for the same tests, but executed in a system with the concurrent reference counter algorithm of Section 3. The meaning of the columns is the same.

Benchmark	alloc	scan_sa	mark_red	scan_green	collect	time(s)
acker	253175	9304	57966	57434	532	0,025
conctwice	42834	897	7210	7127	83	0,0038
fiblista	14837640	35112	956874	935527	21347	2,732
recfat	335959	11421	112663	112075	588	0,0702
somamap	94309	1647	11896	11773	123	0,0298
somatorio	35298	532	6052	5980	72	0,0208

Table 2: Results for the execution of six test programs with concurrent reference counting

The two tables show that the number of calls for memory management functions is greater in the concurrent architecture. This happens because the collector calls `scan_status_anlyzer` every time the increment and decrement queues are both

empty, while the sequential version only examines the status analyzer whenever there are no free cells [Lins, 2002]. However, overall execution times where, on average, about 20% smaller for the concurrent reference counting system, in relation to the sequential version. Albeit these performance results are preliminary – for example, the total execution time of the benchmark programs above is still too small – they indicate that the concurrent architecture of Section 3 represents a performance gain to programs that use it, without any need to alter the source code of user programs; besides, such a garbage collector uses better the available hardware in multi-processor systems, even when the executing programs are not concurrent themselves.

6 Conclusions and lines for further works

This paper presented a new architecture for concurrent lazy cyclic reference counting on multi-processor systems, targeted for implementation on current commercial hardware and with less need for explicit synchronization than its predecessors. This architecture was implemented in a 1.6GHz Athlon MP computer with 2 processors and 512Mb of RAM. A lazy functional compiler was developed, together with the garbage collection subsystem, and its performance was tested against the sequential version of the same algorithm. Six benchmarks were used to test the performance of the architecture proposed. A rise in throughput of about 20% was observed between the sequential and the concurrent version for the programs tested.

The implementation of the language and the garbage collector serves as a test bed for further and more complex benchmarks as well as allowing for the possibility of testing different garbage collection strategies. Considering that the architecture proposed in this paper was based on a previous proposal by Lins [Lins, 2005], a comparison between both versions would be interesting to verify the real gains achieved after minimizing the amount of required synchronization. However, Lins' architecture was never implemented; this is planned for future works. Another direction for further investigation is the use of more computer intensive benchmark programs – preferably real-world programs – that could provide more realistic figures on how the proposed algorithms would behave in production situations. In a recent paper Lins and Carvalho Jr. [Lins and Carvalho, 2007] introduce the notion of "permanent" or "tenured" objects to cyclic reference counting, making the local mark-scan more efficient. The extension of the architecture presented herein to encompass such objects is on progress.

The source code for the benchmarks, compiler and garbage collector may be found at: http://postele.homelinux.net/~andrei/faul.tar.gz.

References

[Andrews, 1999] Andrews, G. R.: "Foundations of Multithreaded, Parallel, and Distributed Programming", Addison-Wesley Professional (1999)

[Bacon et al., 2001] Bacon, D. F., Attanasio, C. R., Lee, H. B., Rajan, R. T., Smith, S.: "Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector"; Proc. of the SIGPLAN Conference on Programming Language Design and Implementation, ACM Press (2001)

[Collins, 1960] Collins, G. E.: "A method for overlapping and erasure of lists"; CACM (Communications of the ACM), 3, 12 (1960), 655-657.

[Dijkstra et al., 1976] Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., Steffens, E. F. M.: "On-the-fly garbage collection: An exercise in cooperation"; Lecture Notes in Computer Science, No. 46. Springer-Verlag (1976)

[Fenichel and Yochelson, 1969] Fenichel, R. and Yochelson, J.: "A Lisp garbage collector for virtual memory computer systems"; CACM (Communications of the ACM), 12, 11 (1969), 611-612

[Jones and Lins, 1996] Jones, R. and Lins, R. D.: "Garbage Collection: Algorithms for Automatic Dynamic Memory Management", John Wiley & Sons ltd. (1996)

[Levanoni and Petrank, 2006] Levanoni, Y. and Petrank, E.: "An on-the-fly reference counting garbage collector for Java"; ACM TOPLAS (Transactions on Programming Languages and Systems), 28, 1 (2006), 1-69

[Lins, 1991] Lins, R. D.: "A shared memory architecture for parallel cyclic reference counting"; Microprocessing and Microprogramming, 34, 31-35 (1991)

[Lins, 1992a] Lins, R. D.: "Cyclic reference counting with lazy mark-scan"; IPL (Information Processing Letters), 44, 4 (1992), 215-220

[Lins, 1992b] Lins, R. D.: "A multi-processor shared memory architecture for parallel cyclic reference counting"; Microprocessing and Microprogramming, 35, 563-568 (1992)

[Lins, 2002] Lins, R. D.: "An efficient algorithm for cyclic reference counting"; IPL (Information Processing Letters), 83, 3 (2002), 145-150

[Lins, 2005] Lins, R. D.: "A new multi-processor architecture for parallel cyclic reference counting"; SBAC-PAD '05: Proceedings of the 17th International on Computer Architecture on High Performance Computing, pp. 35-43, IEEE Press (2005)

[Lins and Carvalho, 2007] Lins, R. D. and Carvalho Jr, F. H. de: "Cyclic reference counting with permanent objects", in this volume

[Martinez et al., 1990] Martinez, A. D., Wachenchauzer, R., Lins, R. D.: "Cyclic reference counting with local mark-scan"; IPL (Information Processing Letters), 34, 31-35 (1990)

[McBeth, 1963] McBeth, J. H.: "On the reference counter method"; CACM (Communications of the ACM), 6, 9 (1963), 575

[McCarthy, 1960] McCarthy, J.: "Recursive functions of symbolic expressions and their computation by machine"; CACM (Communications of the ACM), 3, 3 (1960), 184-195

[Peyton-Jones, 1987] Peyton-Jones, S. L.: "The Implementation of Functional Programming Languages", Prentice-Hall International (1987)

[Valois, 1994] Valois, J. D.: "Implementing lock-free queues"; Proc. of the Seventh International Conference on Parallel and Distributed Computing Systems, pp. 64-69 (1994)

[Turner, 1979] Turner, D.A.: "A new implementation technique for applicative languages"; Software Practice and Experience, 9, 31-49 (1979)