# Incremental Distributed Garbage Collection using Reverse Reference Tracking

M. Schoettner, R. Goeckelmann, S. Frenz, M. Fakler, P. Schulthess

University of Ulm, Computer Science Faculty, 89069 Ulm Germany
michael.schoettner@uni-ulm.de

**Abstract.** Most modern middleware systems like Java Beans and .NET provide automatic garbage collection (GC). In spite of the many distributed solutions proposed in literature collection is typically limited to a single node and simple leasing techniques are used for remote references. In this paper we present a new incremental multistage GC. It has been implemented in the Plurix operating system but might easily be applied to other platforms. The scheme works incrementally and avoids blocking remote nodes. The reverse reference tracking scheme efficiently detects acyclic garbage and is also used for finding cyclic garbage without precomputing a global root set. To minimize network communication cycle detection splits into a local and a global detection part. Keeping the object markers in a separate stack avoids invalidation of replicated objects. Performance measurements show that the proposed distributed GC scheme scales very nicely.

## 1 Introduction

Garbage Collection (GC) relieves the programmer of explicit memory management and avoids memory leaks and dangling pointers. This is important on a single node system and almost indispensable in a distributed and persistent environment. As a consequence most modern middleware systems such as Java Beans and .NET provide automatic GC. These commercial GCs are typically based on scanning algorithms (mark and sweep) for a single node and fall back to a leasing scheme for remote references in distributed programs. In the literature numerous more sophisticated distributed GCs have been proposed [6].

Efficient GC for a distributed environment is more of a challenge than for a single machine. Basic scanning algorithms can not detect concurrent manipulation of pointers during the execution of the GC task and require suspending all other execution. Unfortunately in a distributed environment this means stopping all processing in the cluster. Incremental GC algorithms solve this problem, but often require read or write barriers and introduce programmed synchronization between the nodes in the cluster. Furthermore, in a distributed system all changes to objects including those introduced by the GC (e.g. temporary markers) must be propagated to remote object replicas. Hence small changes made on a single node may affect the entire cluster and decrease overall cluster performance.

In this paper we propose a reverse reference tracking scheme to collect incrementally all types of garbage – local or remote, cyclic or acyclic. Objects which are no longer referenced are called acyclic garbage. Garbage cycles consist of at least two objects referencing each other but neither of these objects is referenced from the root set.

The acyclic GC phase is a simple reference counting scheme with local and global parts of the computation. Unlike other scanning algorithms our reverse reference tracking avoids an atomic precomputation of the global root set and scales smoothly to larger clusters. The second phase collects cyclic garbage in an incremental fashion. Invalidation of remote replicates is avoided by storing the temporary marks separate from the candidate objects in small tables.

The remainder of the paper is organized as follows. In section 2 we briefly present relevant parts of the Plurix architecture followed by a discussion of related work in section 3. In section 4 we present our GC scheme which uses reverse reference tracking. Subsequently, we present the measurement results indicating the scalability of the proposed approach. The conclusions and an outlook on future work is given in the last section 6.

## 2 Plurix Architecture Aspects

Plurix is a native cluster operating system (OS) which simplifies distributed and parallel programming [3]. The entire OS is written in Java (with some minor language extensions for device drivers) and works in a type safe and object-oriented language framework continuing the OS development which was convincingly demonstrated by the Oberon system [2].

Distributed Shared Memory (DSM) in Plurix offers an elegant solution for distributing and sharing data in a cluster of loosely coupled PCs [8]. Applications running on top of the DSM are unaware of the physical location of objects. Remote objects are automatically transferred to an accessing node by the runtime system. Plurix implements a distributed heap (DHS) on top of the DSM which hosts language objects, kernel objects, code segments and device drivers.

Tracking references to objects is a requirement both for the GC scheme and for the object relocation facility. The latter is needed to compact the heap, to resolve false sharing (page thrashing) situations, and to support type evolution. We have developed the so-called *backpack* scheme to track all references to an object. The basic idea is that each object can track up to three references in its own header accommodating the majority of all reference situations (inline backlinks). The reverse tracking links are called *backlinks*. If more than three references are tracked *backpacks* are created on demand. These are separate hash tables containing additional backlinks. A detailed description of backlinks and backpacks can be found in [1].

Any heap object may be registered and then looked up in the directories and subdirectories of a cluster-wide name service. This corresponds to the directory structure of traditional file systems but the functionality of the name service is extended to store symbol tables, configuration information, and to cover all naming issues occurring in the OS. Any heap object reachable from the name service root is not garbage and thus persistent.

The Plurix DHS detects memory access using the Memory Management Unit (MMU) of the CPU thus implementing a page-based DSM. Since individual pages and the allocated objects get replicated a distributed consistency protocol is necessary. Plurix uses a strong consistency model, called transactional consistency [3]. All actions in Plurix regarding the DHS are encapsulated in restartable transactions (TA) combined with an optimistic synchronization scheme. Before a page is modified by a TA the OS creates a shadow image. During the commit phase the addresses of all modified pages are multicast and the receiving nodes will invalidate these pages. Those nodes that detect a collision, abort themselves voluntarily.

In case of an abort all modified pages in a TA are discarded. Shadow images are used to reset the DHS is to the state before this conflicting TA. A token mechanism guarantees that only one node at a time can enter the commit phase. Currently, the token is passed according to a first wins strategy, but improved fairness strategies are currently being investigated. For a more detailed discussion about consistency management, fault tolerance, and persistence see [3].

## 3 Related Work

In this section we briefly discuss GC algorithms which were designed for distributed environments or whose ideas inspired our implementation. An excellent summary of basic GC algorithms is found in [6].

**Copying Algorithms**

These schemes copy all live objects (reachable from the root object set) from one part of the address space to another and the garbage objects are left in the source portion. After the "copy" action heap fragmentation is eliminated but copying many small objects (even if only logically) may be time consuming and expensive invalidations of live remote objects are unavoidable.

LeSergent and Berthomieu [5] have developed an copying algorithm for a distributed GC. Each process in the system has a uniform view of the DSM. The memory is divided into parts with equal size, e.g. physical pages. A single page may be dynamically assigned to one or more processes at a time. If a process tries to access a page which is not present the page is fetched across the network and locally assigned. For this algorithm it is necessary to lock pages if a process needs write access to it. As a consequence nodes may be blocked during the GC cycle.

**Mark-and-Sweep Algorithms**

These algorithms mark each object reachable from the root set. Unmarked objects are garbage. Setting marks within an object may lead to many invalidations of remote objects. It is preferable to store marks outside of the objects, e.g. in bitmap- or hash-tables. Hash-tables consume less physical memory than the bitmap approach but are still expensive in a scenario with many small objects (e.g. 32-64 byte) that are common in object-oriented languages.

A mark-and-sweep algorithm for a distributed system was developed by Yu and Cox [10] in 1996. They designed a GC scheme for the Treadmarks DSM system [4]. Here the heap is divided into blocks in which each process can allocate its own objects. After allocation, the process gains ownership of the object. Objects can be either "local" meaning that the process is owner of this object or "remote". "local" objects which are used by other nodes are marked as "exported". Remotely owned objects are "imported". Both kinds of objects are tracked using import-/export tables. References to "remote" objects are handled using weighted reference counters without using indirection objects i.e. the weight of an object may be less than the weight of all references to it. The GC itself is divided into a local and a global part. The local part is a mark-and-sweep scheme examining entries in the export table, but it is unable to detect distributed cyclic garbage. The global GC part will stop the cluster. All objects reachable from a local root are marked; references to other nodes are recorded and afterwards sent to the associated node, which continuous marking. These steps are repeated until no more references to other nodes exist.

### Reference Counting Algorithms

These GC algorithms depend on a counter for each object, recording the number of existing references. The placement of the reference counter raises a problem similar to the placing of the marks of a mark-and-sweep algorithm. Although the GC is simple and does not block the cluster it cannot detect cyclic garbage without special provisions. Detecting cyclic garbage mostly depends on marking algorithms or removing internal counts (i.e. the reference counter is decremented for each pointer which potentially references another object from the same garbage cycle) [11]. This modifies all checked objects and hereby causes unwarranted invalidations.

Traditionally, the reference counter is included in the object and this forces a modification of the object each time a reference to it is created or destroyed. Invalidation of an object during the creation of a reference can be avoided by using weighted reference counting. But objects can not always be identified as garbage and are modified when a reference is deleted.

Reference counting GC faces additional problems if a node crashes. In this case references to an object are lost but the reference counter is not decremented. Now the reference counter never reaches 0 and the object will not be collected.

David Bacon [11] has presented a GC strategy which is based on reference counting but also collects cyclic garbage. In a separate structure (a so called RootBuffer) the algorithm remembers all objects which could potentially be cyclic garbage. Separate from the traditional reference counting mechanism, the GC scheme contains a second phase in which cyclic garbage is detected traversing all reachable objects starting with the objects included in the RootBuffer. During this computation the reference counter of reached objects is decremented to remove internal reference counts (references which points from one potentially cyclic garbage object to another), and the objects are marked. The algorithm is able to collect cyclic garbage in linear time but it needs to modify the traversed objects. Objects are cyclic garbage candidates if their reference counter is decremented but does not reach zero and is not incremented before the cyclic detection part of the GC is started. This condition may be true for many live objects leading to a large number of invalidations of replicated objects.

### Algorithms basing on an Inverse Reference Graph

The first GC depending on the inverse reference graph was made in 1991 by Piquer [13]. The algorithm uses Indirect Reference Counting based on a diffusion tree which eliminates the need for increment and decrement messages to adjust the reference counter of an object. This avoids race conditions which can lead to incorrect behavior of distributed reference counters. Shapiro [9] extended this approach. Scion-Stub Pointer Chains uses parent pointers to track where references to an object are located. These pointers build the inverse reference tree from an object to its accessors. The GC works similar to traditional reference counting and is not able to collect cyclic garbage.

A similar approach was presented by Birrell [7]. The algorithm eliminates the reference chains by maintaining a set of identifiers for processes with references to an object. To determine this ID-set the transfers of object references to another process are handled by a remote procedure call. Premature collection of objects is prevented by forcing the sender of a reference to keep its copy until receipt is verified.

Another GC strategy depending on the inverse reference graph was presented by Matthew Fuchs [12]. The described algorithm solves the problem of discovering the distributed root set for a mark- and-sweep GC by starting with any object and traversing inverse pointers. The algorithm uses a three color marking to determine whether an object is garbage. An object is live, if the inverse pointer graph contains at least one root object. Root objects are separately marked so that they can be identified. The algorithm is interesting as it can collect garbage without knowledge of the current cluster state and because it is not necessary to know each root object, but it sets marks in shared objects and thus invalidates replicated objects.

## 5 Garbage Collection using Reverse Reference Tracking

Plurix is designed for both distributed and parallel computing but also for cooperative working. Hence its GC must be capable to collect all types of garbage and run concurrently with other applications and without significantly degrading cluster performance. To achieve this goal, the GC should neither utilize excessive network capacity nor block the cluster during execution. The objective of keeping network traffic low requires that write access to objects must be kept to a minimum within the GC, as this would lead to invalidations of replicated objects or of 4 KB pages that could store dozens of objects within Plurix.

**Non-Cyclic Garbage Collection using Reference Counting**

Reference Counting is conceptually simple but in a distributed environment it is important to avoid frequent modification of objects. Piquer [13] has shown that instead of a reference counter backward references can be used, too.

In Plurix the bookkeeping of references is primarily used for relocation of objects but it can also used for GC at little additional cost. We merely count the number of references stored in backlinks within the object itself and in associated backpacks. An object is garbage if all backlink entries from the object are empty. Special root objects which are never garbage are marked by a special non-garbage flag by the OS.

The bookkeeping of references modifies objects only when the "in-line" backlinks are changed. The respective backpack table-object is deleted if the last object reference is removed. The memory management makes sure that backpacks do not co-reside with normal objects on a page, i.e. aborts of other TAs may only occur if both TAs try to modify a reference to an object.

Reference counting schemes also need to consider stack references and CPU registers. Because of the transactional processing in Plurix this can be done elegantly. The GC runs as a separate TA thus seeing only committed and valid state of objects. Most TAs (e.g. processing an event) commit with an empty stack and with empty registers. Some TAs (e.g. for parallel computing) may commit with a non-empty stack that is consolidated during commit time including CPU registers - all references on the stack are recorded in backpacks during commit time. Postponing stack reference tracking is recommended because not all applications need this feature. Often the stack shrinks before commit and only the references from a small residual stack need to be tracked and only once during commit.

Plurix will find all objects in the heap by stepping from one object to the next. The reference counting algorithm can run concurrently on several nodes. The GC only has to check objects which are present locally, as each object must be present on at least one node. The backlinks of each such object are checked, and it is collected in case of garbage.

Acyclic GC can be run without causing additional network traffic during detection of garbage objects, as only local objects are inspected and the internal backlinks contain sufficient information about the state of an object. Network traffic and collisions only occur if a garbage object and all its references to remote objects are deleted.

**Cyclic Garbage Collection using Inverse References**

The major challenge for a GC in a distributed system is to detect and collect cyclic garbage. After collecting non-cyclic garbage the remaining objects are either alive or part of cyclic garbage. Cyclic GC is used to break the cyclic structure of garbage objects so that these objects can be collected during the next execution of the non-cyclic GC. We have developed an incremental variation of mark-and-sweep to detect cyclic garbage. The marks are kept outside the objects to avoid invalidations. Backpacks provide all information for inverse reference tracking.

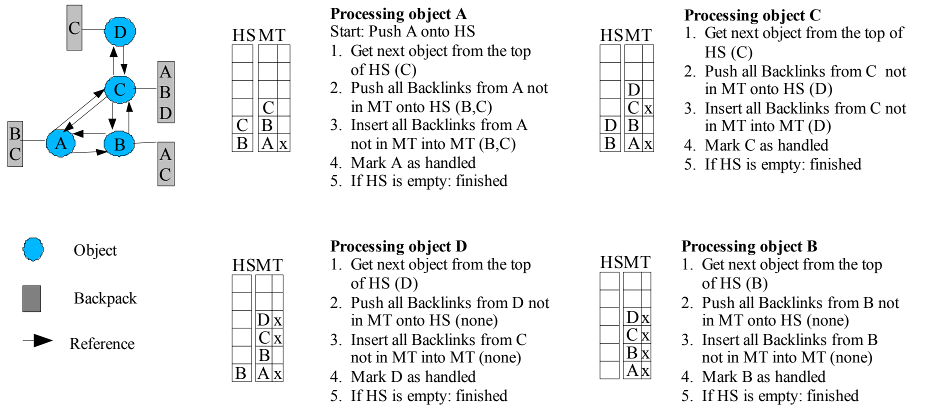
In traditional systems the set of root objects must be determined by obtaining the root subset from each node or running the GC simultaneously on each node. More easily our algorithm starts at an arbitrary object which is locally present and traverses the inverse reference graph searching for a root object. If none is found, the object is part of cyclic garbage and should be deleted. Thereby all references included in this object are removed. Other objects which were traversed during this GC scan are not yet collected because the remaining members of this cycle will be detected by the non-cyclic GC if the cycle is broken at an appropriate place. Otherwise the cyclic GC will identify the next candidate and so forth.

It is necessary to mark each traversed object to avoid endless loops during the execution of the GC. These marks must not be located inside the objects to reduce invalidations. Unlike traditional mark-and-sweep algorithms not all objects in the cluster need to be marked, therefore it is possible to place the marks in a separate "marking table" (MT, hashed or otherwise). As all objects in Plurix are located on 4 Byte borders, the least significant 2 Bits of each address or backlink are 0. These bits in the MT are conveniently used to remember whether an object has already been checked. In addition to the MT there is another table (i.e. an integer array) used during the GC which is organized as a stack. All encountered backlinks which are not already checked are placed on top of this handle stack (HS).

At the start of a GC TA the MT and HS are created unless older tables can be reused. Both tables are not shared so that TAs on other nodes are not affected by modifications of these tables. Additionally, the memory management allocates the HT and MS on a 4kB border and with a size of a multiple of 4 KB to avoid false sharing so that modifications do not cause invalidations of unconcerned objects. The size of both tables is limited by a configurable value, given to the GC TA at start time. This defines the maximum depth of cycles which might be detected by this TA but does not reduce the capability of the algorithm. If the GC is terminated due to an exhausted MT or MS, the GC can be restarted with a larger one. In contrast to the tables needed for general mark-and-sweep, the tables for cyclic garbage detection are very small. The GC has successfully detected a cyclic structure if the HS is empty and no root object has been found. In this case at least the object at which the GC has been started should be deallocated. References from this object to another one are deleted. Depending on the remaining time, other objects in the MT may be deleted since they are not reachable from the root set.

The steps of the algorithm are described below and an example is shown in fig. 1:

1. The flag field of the object is checked. If it is marked as non-garbage the GC terminates because the object is a root object. The chosen object is reachable and not garbage.
2. The address of this object is inserted into MT. If the MT is exhausted go to step 7.
3. All backlinks of the object are inserted into the MT and pushed onto the HS; duplicates are ignored. If the MT or HS is exhausted go to step 7.
4. The MT entry for the object is marked. This object is now completely handled.
5. If the HS is not empty get next address of an object from the HS and go to step1.
6. If the HS is empty, the chosen object is part of cyclic garbage and can be deleted. MT and HS can be cleared and the algorithm will terminate.
7. The GC terminates without being able to detect a root object. The chosen object is treated to be non-garbage.



| HS | MT | |
|---|---|---|
| | | |
| | | |
| | C | |
| C | B | |
| B | A | x |

**Processing object A**
Start: Push A onto HS
1. Get next object from the top of HS (C)
2. Push all Backlinks from A not in MT onto HS (B,C)
3. Insert all Backlinks from A not in MT into MT (B,C)
4. Mark A as handled
5. If HS is empty: finished

| HS | MT | |
|---|---|---|
| | | |
| | D | |
| | C | x |
| D | B | |
| B | A | x |

**Processing object C**
1. Get next object from the top of HS (C)
2. Push all Backlinks from C not in MT onto HS (D)
3. Insert all Backlinks from C not in MT into MT (D)
4. Mark C as handled
5. If HS is empty: finished

| HS | MT | |
|---|---|---|
| | | |
| | | |
| | D | x |
| | C | x |
| | B | |
| B | A | x |

**Processing object D**
1. Get next object from the top of HS (D)
2. Push all Backlinks from D not in MT onto HS (none)
3. Insert all Backlinks from C not in MT into MT (none)
4. Mark D as handled
5. If HS is empty: finished

| HS | MT | |
|---|---|---|
| | | |
| | | |
| | D | x |
| | C | x |
| | B | x |
| | A | x |

**Processing object B**
1. Get next object from the top of HS (B)
2. Push all Backlinks from B not in MT onto HS (none)
3. Insert all Backlinks from B not in MT into MT (none)
4. Mark B as handled
5. If HS is empty: finished

**Fig. 1.** Cyclic Garbage Detection Example

Because cyclic garbage detection can be a time consuming operation depending on the size of the cycle and the distribution of the affected objects, the cycle GC comes in two variants: local and global cyclic GC. Both variants may be aborted at any time without affecting the cluster state. Which variant and which parameters of the cyclic GC are started is configurable reflecting CPU load of the node, network load, and low memory, etc.

The local part of the cyclic garbage detection checks only those objects which are locally present. This can be determined by the flags (set by the MMU) in the page tables. For each candidate object the backpack or respectively the backlinks are inspected and the inverse reference tree is built. As soon as a backlink references a remote object the cyclic garbage detection stops and the object is regarded to be live. The GC will choose the next candidate object until the configured time slot if any expires. Since even in a distributed environment many objects are locally used the local phase is useful – effectively reducing network traffic.

The second part of the cycle detection GC works on the entire cluster. Again objects which are locally present are used as a start for cycle detection but all backlinks are checked. To reduce network traffic, the cluster wide cycle detection algorithm tries to detect a local root object before remote objects are transferred to the node. Remote pages are not requested until all local references are checked and no root object was yet found, hence the GC does not cause network traffic for objects which are reachable from the local root subset. To distinguish between local and remote the inverse reference stack of the GC is duplicated. One stack is used for objects locally present and the other for remote ones. When the local stack is exhausted and no root object was found, the backlinks preserved in the remote stack are inspected. If such a remote object contains backlinks to locally present objects these are checked before other remote backlinks are observed. This ensures, that remote pages are only requested if it is inevitable. In most cases a local root object is found, if the chosen object is not part of cyclic garbage, before all remote backlinks have been checked.

## 6 Measurements

Measurements were made on a cluster of 16 nodes (AthlonXP 2500+ with 512 MB RAM) using a switched FastEthernet. We compare our GC with a traditional blocking mark-and-sweep GC (BMSGC). Since blocking GCs are faster than the corresponding incremental solutions the execution time of BMSGC can be viewed as the lower bound. In the first part the measurements only use a single node. We allocated 13'800 objects whereof 1'600 were acyclic and 1'200 cyclic garbage. The cyclic garbage was spread over 36 cycles each containing between two and eight objects. Table 1 shows the execution times of different steps of the Plurix GC (PGC) and for the BMSGC. Times shown are an average of 10 independent runs.

**Table 1.** Execution times of PGC and BMSGC on a single node

| Action | exec. time (ms) | #objects processed |
|---|---:|---:|
| PGC: acyclic garbage | 6,03 | 2'653'000 |
| PGC: cyclic garbage (detection only) | 54,00 | 252'000 |
| PGC: cycle detection & removal | 55,00 | 251'000 |
| BMSGC: remove marks | 3,27 | 4'220'000 |
| BMSGC: mark phase | 18,07 | 763'600 |
| BMSGC: delete objects | 4,12 | 3'349'000 |

The measurements show that the detection of acyclic garbage is much faster in PGC than in BMSGC whereas the situation is reversed for cyclic garbage on a single node. But BMSGC requires marking all objects each time the GC is called to determine if an object is garbage or not - and of course it represents a lower execution bound if we can afford to block all nodes.

In the second part we evaluated the performance of PGC in cluster operation. For these measurements we have allocated 61'600 objects whereof 12'000 are acyclic and 9'600 cyclic garbage. The latter included 4'000 objects having references to remote nodes. As the acyclic GC stage is able to check individual objects it can be executed concurrently on all nodes. Inter-node communication is necessary only if an object with a reference to a remote object is deleted because this requires deleting the associated backlink on the remote node. In this case the object deletion is increased by 784 μs - reflecting network latency. Of course this will be less significant for faster networks. In the best case there are no remote references and the acyclic GC will scale almost linearly with the number of nodes.

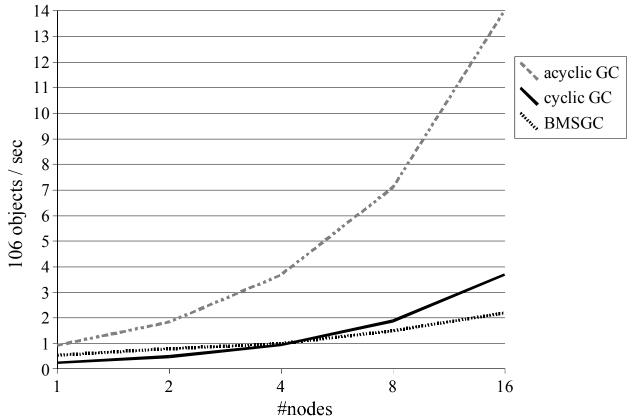


**Fig. 2.** Performance in cluster operation PGC & BMSGC

The performance of the cyclic garbage detection in cluster mode depends on the number of objects that need to be checked and the number of page requests to remote objects to be performed. If all local objects are referenced by some object that is part of the root set, no network communication is necessary. In that best case the scalability of the GC depends only on the distribution of objects in the cluster. For the applications we use (distributed and parallel ones) this is true for approximately 90% of all objects. This is the reason why PGC outperforms BMSGC in the cycle detection in cluster operation. Fig. 2 shows the scalability for the acyclic and cyclic stages of PGC and BMSGC. The throughput given in $10^6$ objects per second has been computed using measurements with the example with 61’600 objects described above. Although for this example BMSGC offers a better performance on a single node PGC outperforms the BMSGC for four and more nodes and scales quite well. As we assume concurrent execution of PGC on all nodes periodically this is a very nice result.

Further measurements might be beneficial and we do not claim that there won’t be a special case where the one or other sophisticated GC will be faster than the one we propose. But generally speaking we find that our approach scales well in a distributed DSM environment and that it is an interesting option for other distributed scenarios as well.

## 7 Conclusion

In this paper we have proposed an incremental multistage GC built on reverse reference tracking and keeping the reverse references in so-called backpack/backlinks. The proposed GC approach is easily be adapted to other distributed systems and does not limit the GC to a special environment. The first stage of our GC detects non-cyclic garbage and is basically a reference counting GC evolving directly from the backpack concept and scaling very nicely.

The cyclic phases deal with cyclic garbage and can be executed concurrently without blocking the cluster. There is one stage only working on local objects (avoiding network traffic) and a second stage working at the cluster level if necessary. Marks are stored outside objects in small tables avoiding invalidations of remote replicas. Furthermore, computation of the global root set and contacting all nodes is not required because of the reverse reference tracking scheme.

The GC algorithm is used in our Plurix OS and has been successfully tested in a cluster with 16 nodes concurrently running distributed and parallel applications. In the future we plan to study different types of applications and to develop heuristics to find good candidates as a starting point of the cyclic GC.

## References

1. R. Goeckelmann, S. Frenz, M. Schoettner, P. Schulthess, “Compiler Support for Reference Tracking in a type-safe DSM“, in: Proc. of the Joint Modular Languages Conf., Klagenfurt, Austria, 2003.
2. N. Wirt and J. Gutknecht, „Project Oberon“, Addison-Wesley, 1992.
3. The Plurix project: www.plurix.de.
4. Amza C., Cox A.L., Drwarkadas S. and Keleher P., „TreadMarks: Shared Memory Computing on Networks of Workstations“, in: Proc. of the Winter 94 Usenix Conference, 1994.
5. T. Le Sergent and B. Berthomieu, “Incremental multi-threaded garbage collection on virtually shared memory architectures“, in: Proc. Int. Workshop on Memory Management, number 637 in Lecture Notes in Computer Science, pages 179-199, Utrecht (NL), 1992.
6. Richard Jones, “Garbage Collection: Algorithms for Automatic Dynamic Memory Management“, JohnWiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.
7. A.Birrell et al. , “Distributed garbage collection for network objects“, in Technical Report 116, DEC Systems Research Center, 1993.
8. K. Li, “IVY: A Shared Virtual Memory System for Parallel Computing”, In Proceedings of the International Conference on Parallel Processing, 1988.
9. M. Shapiro, D. Plainfossé, P. Ferreira, L. Amsaleg, “ Some Key Issues in the Design of Distributed Garbage Collection and References“, in seminar on "Unifying Theory and Practice in Distributed Systems," Dagstuhl Int. Conf. and Res. Center for Comp. Sc., 1994.
10. W. M. Yu and A. L. Cox, “Conservative garbage collection on distributed shared memory systems“, in: Proc. of the Int’l Conf. on Distributed Computing Systems (ICDCS-16), 1996.
11. D. F. Bacon and V. T. Rajan, “Concurrent Cycle Collection in Reference Counting Systems“, Proc. European Conference on Object-Oriented Programming, June 2001, volume 2072 of Lecture Notes in Computer Science, Springer Verlag.
12. M. Fuchs, “Garbage Collection on an Open Network“, in volume 986 of Lecture Notes in Computer Science, 1995.
13. J.M. Piquer, “Indirect Reference Counting, a distributed garbage collection algorithm“ , in: PARLE'91-Parallel Architectures and Languages Europe, volume 505 of Lecture Notes in Computer Science, page 150-165, Eindhoven (NL), June 1991, Springer-Verlag
14. J. M. Piquer, “Indirect Mark and Sweep“, in Baker HG (Ed.), Memory Management, Proc IWMM95 LNCS 986, Springer-Verlag, 268-282.