Comprehensive and Robust Garbage Collection in a Distributed System

Niels Christian Juul & Eric Jul
DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK 2100 Copenhagen Ø, Denmark
Phone: +45 35 32 18 18   Fax: +45 35 32 14 01   E-mail: {ncjuul|eric}@diku.dk

Presented at International Workshop on Memory Management, St. Malo, France, September, 1992
(Published in [Bekkers 92] as [Juul 92]).

Abstract

The overall goal of the Emerald garbage collection scheme is to provide an efficient "on-the-fly" garbage collection in a distributed object-based system that collects *all* garbage, and that is robust to partial failures.

The first goal is to collect *all* garbage in the entire distributed system; we say that the collection is *comprehensive* in contrast to conservative collectors that only collect most garbage. Comprehensiveness is achieved by employing a system-wide mark-and-sweep collection based on concurrently running collectors, one on each node.

The second goal of our collector is to be robust to partial failures. When facing node failures the collector will progress in the available parts of the system and, when necessary, wait for temporarily unavailable nodes to become available again.

The scheme is being implemented on a network of VAXstations at DIKU. The full scheme employs two concurrent mark-and-sweep collectors on each node in the distributed system, one for comprehensiveness, one for expediency. Concurrency is achieved by using an object protection and faulting mechanism.

Keywords: Garbage collection [*mark-and-sweep, faulting, comprehensive*], Distributed systems [*distributed control, termination detection, fault-tolerance*], Concurrency, Object-oriented systems, Robustness, Emerald, Algorithm.

1 Introduction

The first goal of our distributed garbage collection scheme is to collect *all* garbage in a entire distributed system. We have introduce the term *comprehensive collection* to denote such schemes. In contrast, partial or conservative collection is a priori non-comprehensive. In general, the garbage collection problem can be formulated as a graph problem, where the vertices in the graph are objects and each directed arc represents a reference from one object to another. In contrast to a more conservative collection, a comprehensive collection essentially needs to perform a system-wide traversal of the graph in order to identify which objects are still in use and which are garbage. We attain a comprehensive collection by combining a basic mark-and-sweep collection scheme with mechanisms for concurrency and distribution.

Unfortunately, any comprehensive collector in a distributed system will, due to large network overheads, have problems collecting garbage fast enough to keep up with new object allocations. Thus, it is necessary to supplement our collector with an expedient, but conservative collector. In this paper, we concentrate on the issues relating to comprehensive collection.

The second goal is *robustness* to partial failures. In large distributed systems, the probability of failures becomes significant. Thus, we cannot expect the entire system to be available concurrently long enough to complete a comprehensive collection. This has lead us to investigate robust garbage collectors. A robust garbage collection scheme must cope with both short and long term unavailable parts of the system—partly by adapting its behavior to the situation, and partly by compromising on its goals.

While still being comprehensive, the collector must be able to survive during temporary unavailability. The collector must progress in the available parts of the system and wait for the needed, but unavailable, parts to become available again.

When more permanently unavailable parts block the comprehensive collection, both robustness and expediency demand that garbage is collected in the available parts. This may be achieved by another collector that collects garbage in the available parts of the system only. Such a collection cannot be comprehensive as long as the "liveness" of references from unavailable parts is unknown. Based on a conservative estimate of root objects, taking objects potential reachable from unavailable parts into account, this supplementary collector may collect garbage in the available part of the system. By careful selection of the additional root objects, this collection may be nearly comprehensive in the available parts of the system.

The full garbage collection scheme, which is based on at least two collectors on each node, must also reduce the latency introduced into applications. Thus, each collector works concurrently with other processes. The necessary synchronization constraints introduced by this concurrency are achieved by protecting objects, that have not been traversed by the collector, from being mutated by other processes.

Our approach has been to implement such a collection scheme for the Emerald Language [Hutchinson 87b, Raj 91]. Emerald is a distributed, object-based system [Black 86, Black 87, Jul 88b], based on a compiler generating native machine code [Hutchinson 87a] and a run-time system [Jul 88a], designed to take advantage of run-time garbage collection.

The objects handled by the run-time system in Emerald may be *migrated* between the nodes of the distributed system. Immutable objects may be *replicated* instead of moved; the replicas will always have consistent states because their state does not change. To survive node crashes, any object may *checkpoint* its current state to other nodes and/or stable storage. A checkpoint is a passive copy, from which an object may be recovered, if the real object has been lost during a node failure. Emerald is based on reliable inter-node communication, thus only nodes may fail. In our model, nodes are autonomous and have *failed-stop* semantics.

In summary, the Emerald garbage collection scheme does a comprehensive and concurrent collection of all garbage in the distributed system, while being robust to node failures. The full scheme employs two faulting, mark-and-sweep collectors on each node in the distributed system.

Before presenting our comprehensive and robust solution to the distributed garbage collection problem in details, the goals of distributed garbage collection are described (Sect. 2). Based on these, an overview of recently and related work on distributed garbage collection is given (Sect. 3), followed by a sketch of the basic Emerald garbage collection scheme (Sect. 4).

The Emerald solution is detailed in the following sections. First (Sect. 5), we discuss comprehensive garbage detection in a failure-free distributed system, and next (Sect. 6), the expedient collection of local garbage is discussed. Then robustness to failures is added, using distributed control and a distributed termination detection algorithm (Sect. 7). Sect. 8 gives some remarks on storage reclamation, and finally, we summaries our contribution (Sect. 9).

2 Goals in Distributed Garbage Collection

Distributed garbage collection is not only faced with the traditional problems of garbage collection, the very nature of distribution poses further challenges. specifically, robustness to partial failures is a goal that impacts all other goals. We identify the following general goals in garbage collection schemes. The consequences, when taking robustness into consideration, are described in the rightmost column:

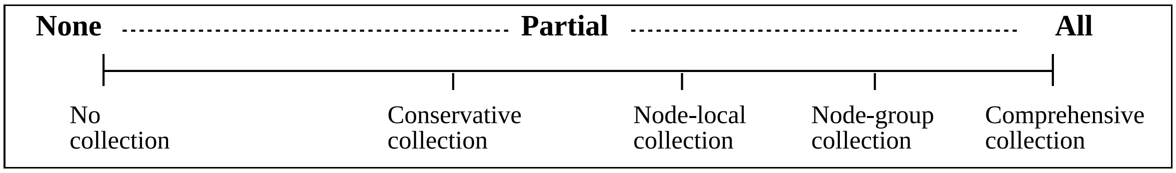

Figure 1: The degree of partial garbage collection

	General goals	**Robustness considerations**
Comprehensiveness	*All* garbage is collected, e.g., no memory leakage.	Continually adapting to the current available parts, while waiting for unavailable parts to become available again as necessary.
Concurrency	The collector and mutators run concurrently on all nodes.	Mutators must not be blocked by a collector due to unavailable parts.
Expediency	Delivery of garbage for recycling in a speed comparable to the speed of new allocation requests.	Collection must complete despite unavailable parts.
Efficiency	Limited overhead per byte of storage collected introduced by each step and the total number of steps needed.	Failures and their circumvention must be handled efficiently. Additional overhead due to robustness must be limited and mainly paid when failures are present.
Correctness	Only garbage must be collected, e.g., no dangling references.	References to unavailable parts and references in checkpointed files must remain valid.

An ideal scheme would fulfill *all* of these goals; unfortunately, this is not possible in general. A comprehensive collection depends on all nodes in the distributed system, thus, the mere presence of communication delays in distributed garbage collection often rules out the possibility of fulfilling the goals of comprehensiveness and expedience by a single collector. Thus, a trade-off between comprehensiveness and expediency is necessary.

By trading off comprehensiveness, we achieve a partial collection, i.e., only part of the garbage is collected. Figure 1 describes the various degrees of partial collection, from collection of nothing, to collection of all garbage, with a broad variety of conservative collectors, which are able to collect only part of the garbage, in between . From the most conservative collectors like Boehm-Weiser [Boehm 88] and Mostly-Copying [Bartlett 88] to comprehensive collectors like the ones implemented for POOL [Augusteijn 87] and Emerald [Juul 93]. In between, we find collectors based on smaller or larger parts of the system with conservative estimates of references from other parts. In distributed systems these may span from one node, over groups of nodes, to nearly all nodes. As a supplementary, expedient collector Emerald employs such a node-local collector on each node. The Galileo and SOR projects are examples of such node-group collectors [Mancini 91, Shapiro 90]. Also [Lang 92] describes distributed collection schemes, which eventually reclaims all inaccessible objects. The partitioning and grouping techniques in the distributed environment, has many similarities to non-distributed techniques like area collection and the partitioning used by generational scavenging.

3 Related Work

Distributed systems like POOL [Augusteijn 87, Beemster 90], Galileo [Mancini 91], Argus [Liskov 86], and those implemented in the SOR project [Shapiro 90, Shapiro 91] all employ garbage collection. These distributed garbage collectors fulfill most of the previous mentioned goals. They do, however, compromise on various aspects of the goals to be able to achieve some of the others.

In general, the non-comprehensive collectors are able to collect more efficiently, while being both expedient and robust to node-failures. The solution described by Shapiro to be implemented in project SOR is robust but may fail to collect all garbage, instead, it has the potential for being expedient. Lang, Queinnec, and Piquer further refine the scheme based on independent node collectors to cooperate for various groups of nodes. The scheme is expected [Lang 92] to be comprehensive under the assumption that each distributed (interconnected) graph of garbage objects is contained in a group of nodes, which do not fail during the collection of the group, i.e., failures during the collection are not tolerated.

The collector for Galileo is implemented as a global stop-and-copy collection of objects on stable storage. If nodes become unavailable a fault-tolerant adaption enables a non-comprehensive collection to complete. The scheme may limit the collection to a subset of nodes and thus only blocking mutators on those nodes. A comprehensive collection is, however, not guaranteed.

A comprehensive collection is often more costly and assumes a simple failure model or no failure at all. The collection scheme for the POOL system is based on global synchronization of node-local mark-and-sweep collectors, and to be comprehensive it depends on complete node availability. The garbage collector for POOL, as described in [Augusteijn 87], does not cope with failures in the distributed system.

The idea of node-local collectors that cooperate has also been used by Liskov and Ladin in Argus. The cooperation is made possible by introducing a logically centralized, but physically replicated, highly available service for cross-node references. Any such reference is registered at the service, which also does cross-node garbage cycle detection. The service is based on synchronized local clocks and bounded delays of cross-node message exchanges.

4 The Emerald Garbage Collection Scheme

Based on the ideas in [Jul 87, Jul 88a, Jul 88b], [Juul 93] describes the design and implementation of a garbage collection scheme for the distributed, object-based system, Emerald, which fulfills all the goals listed in Sect. 2. The solution combines the comprehensiveness of mark-and-sweep collection with distribution and concurrency. The primary methods to achieve concurrency and robustness are distributed control, object protection and faulting, and the use of more than one collector concurrently.

Robustness to partial failures in a distributed system can be achieved when all parts work independently. Still cooperation is needed, but distributed control makes partial failures less threatening to the system. Robustness to partial-failures in distributed systems has lead us to implement our garbage collector without using centralized control.

4.1 The Basic Mark-and-sweep Algorithm

Our basic algorithm is based on a *concurrent* variant of mark-and-sweep garbage collection. The mark-phase is done concurrently with user processes running by *protecting* non-marked objects from being used by the running processes with a *garbage collection fault* mechanism similar to a page-fault mechanism in a virtual memory system [Appel 88, Appel 91]. In Emerald such a protection and faulting mechanism is already available in the implementation of *remote invocation*. Thus, the utilization of the mechanism by the garbage collector is nearly free.

The mark-phase of our basic algorithm uses the traditional three color settings: white, gray, black. Objects are marked either white (potentially garbage), gray (alive with references under consideration), or black (alive with references considered). Furthermore, a root set of objects is given, i.e., the active processes and the "always present" objects.

The mutators may run without problems in the black objects and the algorithm assures that mutators execute in black objects only. They may reference other black or gray objects, but the gray objects are protected. Thus, a mutator, which tries to use a gray object, will be suspended while a fault handler marks it black and traverses it to ensure that the objects, it references, are marked and protected. This way mutators are only faced with black objects. When the collection is started, all objects are white, and all mutators are stopped. Before each mutator is resumed, it is marked black, and its references is marked at least gray. The mark-phase is finished when all gray objects have been traversed and marked black, i.e., when the gray set is empty.

4.2 The Garbage Collection Invariants

The following invariants form the basis of our garbage collection algorithm. A detailed description of the basic algorithm and its implementation is found in [Juul 93]. The general assumption is that *garbage stays garbage*. The collection scheme is based upon this assumption and five invariants.

Invariant 1 (Progression)

During garbage collection objects become darker, never lighter, i.e., shading is a monotone function moving objects from white to gray, and from gray to black.

Invariant 2 (Mutators)

Mutators execute in black objects only.

Invariant 3 (No black-to-white references)

No black-to-white references, i.e., a black object contains references to gray and black objects only.

Invariant 4 (Faulting)

Gray objects are protected, thus any attempt to access a gray object is withhold until the faulting mechanism has changed the object from gray to black (by shading its references at least gray).

Invariant 5 (Termination)

No gray objects indicates that black objects are the surviving ones, whereas the whites are all garbage (and thus reclaimable).

4.3 The Dual Collector Scheme

The Emerald garbage collection scheme consists of two sets of collectors, which are all applied concurrently. The *global* scheme, using one collector on each node in the system, continuously adapts to the current situation and strives to fulfill comprehensiveness while giving up on expediency. The *local* scheme foresees the failures of many parts of the system by performing an independent and expedient, but non-comprehensive, local collection on each node.

The comprehensive collection is achieved by one concurrent mark-and-sweep collector on each node, which cooperate as one global garbage collector across the entire network of Emerald nodes. The set does a comprehensive collection of all garbage, while various parts of the distributed system may be temporarily unavailable. A second set of collectors does an independent, partial collection on each node. These node-local collectors do a more expedient collection of local garbage without being comprehensive. Both sets of collectors proceed simultaneous and in parallel (*on-the-fly*) with the running processes. Each set of collectors adds robustness to the garbage collection scheme. The global collection by waiting for needed but unavailable nodes to become available again while progressing the collection in the available parts of the system. Whereas each local collector is able to collect local garbage while the rest of the system is unavailable. This further adds efficiency and expedience to the scheme, as most objects tend to be short lived and local [Lieberman 83, Schelvis 88, Jul 88b, Rudalics 86].

5 Comprehensive Garbage Collection

In terms of a graph, a comprehensive collection must partition the distributed graph of objects, connected by references, in two very well-defined parts. One containing *exactly all* the objects reachable from the distributed root set, and another containing the rest, i.e., *all* the garbage. During a comprehensive garbage collection, the graph must be traversed from the root set to identify the reachable objects, i.e., the closure of the root set. To ensure the collection of all garbage, the references in the root set and the references inside the objects must be identified exactly.

In general, any traversing algorithm has this property. Thus, the basic algorithm, even in the distributed case, can be based on either mark-and-sweep or copying collection. With focus on garbage *detection*, which is the harder part of the problem in the distributed case, the mark-and-sweep algorithm has been chosen due to its nice separation of garbage detection from garbage reclamation. Our current implementation pays little attention to compaction and locality of references. Though copying collectors may waste up to half of the available memory, they might be considered in future implementations where compaction is combined with object mobility.

In the comprehensive garbage collection scheme in Emerald any node may take initiative to a new *global collection cycle* and inform the other nodes in the distributed system about the decision. Each collector progress on its own node by initiating the collection and doing the marking. References to non-resident objects must, however, be treated differently. To the mutators on the node, we pretend that the non-resident objects are already black, while we accumulate references to them in the *non-resident gray set*. When the gray set of resident objects has been emptied, the non-resident objects are handled by sending a shade request to the node hosting the object. Meanwhile, remote requests to shade objects resident on our node are handled by putting these references in our gray set of resident objects. Each shade request is acknowledged by the node hosting the object, to let the requesting node remove the reference from the non-resident gray set. Thus, a gray reference will stay in the non-resident gray set until the node hosting the object guarantees that the object is at least gray, i.e., gray or black.

The mark-phase is finished when both gray sets are empty on all nodes. This global state is stable, in contrast to the state *both gray sets empty* on a single node. The global state is detected by a two-phase commit protocol. For simplicity, the global termination detection could be detected by approving a coordinator node. The current solution is, however, prepared for robustness to partial failures.

The cooperating collectors, constituting the global collection, may run very independent on each node. They only need to coordinate their actions on three topics:

1. When to start, i.e., when mutators must be stopped and the local part of the distributed set of root objects constructed.

2. During the mark-phase, i.e., when a non-resident object is shaded by requesting the node hosting the object and acknowledging the action back to the requesting node.

3. To determine when the mark-phase is finished, a distributed termination detection protocol must detect the *all gray set is empty* situation.

All nodes may decide to initiate a new cycle of the comprehensive collection and let this knowledge sieve to the other nodes. By adding the information (the cycle number) about a progressing collector in all inter-node messages, any node will become aware of the situation before it engages in the transfer of objects or references with the started nodes. Though the garbage collectors may start at different wall clock time, we are able to identify a common logical clock where no collector were started before and all were started after.

6 A Dual Node-Local Garbage Collector

Due to both expediency and robustness the set of global comprehensive collectors has been extended with another set of independent local collectors.

The local collectors provide expediency by not depending on inter-node communication, and robustness by only using available nodes. We have chosen not to add a node-group collector scheme as a third, intermediate scheme. We find the clustering of nodes irrelevant to our current testbed of only a dozen nodes, to pay the additional cost of maintaining tables of incoming and outgoing references from each node. Furthermore, the two collector scheme is enough to fulfill our goals, thus a third scheme would not add substantially benefits.

The node-local collector is conservative in its definition of the root set, but not in its identification of references between objects. The conservative approach is acceptable, as this collector is supplementary to the comprehensive scheme. The node-local collector will collect local garbage only, i.e., garbage which has never been reachable from other nodes.

The implementation is fairly simple. It takes advantage of a general mechanism that marks objects potentially reachable from other nodes as *ReferenceGivenOut*. When a reference to a resident object is exported to another node, the object is marked as known from outside. Though that reference may later be dropped, the object stays marked during the rest of its lifetime. These marked objects are added to the root set of the node-local collector. When this collector finds references to non-resident objects, they are simply skipped.

Beside the extended root set and the missing needs to communicate with other nodes, the local collector uses exactly the same algorithm as does the global collector on each node. The two collectors on each node need, however, to synchronize, as they are working on the same data. Thus, they have their own data structures and mark fields for color information. To prevent either of them from reclaiming objects, later needed by the other, the two collectors may not have a non-empty set of resident gray objects concurrently. On each node this is achieved by only starting the local collector when the global has an empty resident gray set, which is achievable without communication delays, and by only starting the global collector between the end of the mark-phase and the start of the next local collection.

7 Robust Garbage Collection

The comprehensive global garbage collection scheme is threatened by node failures, as these influence the global state as well as the individual collectors.

The failure model is *failed-stop*, thus, by keeping its main state on stable storage, the collector on each node may always restart in a globally well-defined state. As an aside, a node failure is by itself an effective garbage collection, as a restarting node gets all storage reclaimed, except checkpointed objects. These are saved, e.g., on stable storage, and recovered after the failure. This means that live references may reside on nodes currently unavailable due to a failure.

From a global point of view, node failures may influence the garbage collection as follows:

Before a garbage collection No harm.

During the start of a new collection Nodes may become out of step concerning the global garbage collection state.

During the mark-phase The global invariants about colors and mutators may be broken.

Finishing the mark-phase The global termination detection needs reliable information about all nodes.

During the sweep-phase No harm.

Robustness to node failures, i.e., to partial failures of the distributed system, must take the above situations into account. The problems occur exactly in the situations where the collectors on each node need to coordinate their action (see the listing of the very same three points in Sect. 5).

7.1 Starting a New Collection

It is fairly easy to assure that restarted nodes adapt to the global situation. The presented mechanism to ensure synchronization by tagging all inter-node messages, will also force a restarted node to enter the same garbage collection cycle before it engages in mutating the object graph. If it keeps running locally only, it may, however, not be aware of the progressing global garbage collection on the other nodes, until one of these sends out shade requests or tries to detect global termination. This will eventually happens, and when it does, the restarted node may immediately adapt to the situation, without breaking the global invariants. From a global point of view, all actions, done by the restarted node until then, have taken place before the logical global clock of the start of the mark-phase.

7.2 A Robust Mark-Phase

During the mark-phase, node failures have several impacts. Both while a node is failed and when it is recovered.

When recovering, a node will restart its collection, but come up with references to non-resident objects. It will need to send out shade requests for these references, even though it might have done so before the failure occurred. Shading is an idempotent function, thus no invariant is broken, only performance is degraded (but this is insignificant compared to the node crash and reboot sequence). The scheme also covers the cases where a shade request has been sent but the reply was lost. For better performance, a node may save its received acknowledgements to remote shade requests on stable storage and use this information when recovered.

While a node is unavailable, other nodes may have references to objects on it. They cannot shade these objects until the node is available again. For simplicity, we have implemented the shade request mechanism as a repeated broadcast with exponentially back-off. Thus, non-acknowledged requests will be sent out until they are eventually acknowledged.

7.3 Distributed Termination Detection

The detection of the global state *all gray set are empty* can be difficult when nodes may fail independently and randomly often. To achieve comprehensiveness we must ensure that all nodes have finished their mark-phase and that no request is in transit.

The latter is ensured by the acknowledgement of shade requests. The references in the non-resident gray set are kept there until an acknowledgement is received by the shade reply mechanism (Sect. 5).

The two-phase commit protocol can be started by any node. A node may do so when it believes that the collection is done. Such a decision is based on its own status and the network traffic. More precisely, both gray sets of the node must be empty, and no nodes must have been broadcasting shade requests for a while. The termination detection protocol is robust to temporary node failures; it only depends on nodes being pairwise available. The current implementation depends on each node being aware of all other nodes in the system. This assumption holds in the current Emerald prototype. A system with a very large number of nodes should use another protocol.

8 Storage Reclamation

Though we emphasis on garbage detection, a short presentation of the reclamation part of our garbage collection scheme is given here to complete the picture.

The sweep-phase of all our collectors (both local and global) has been relinquished from the mark-phase in a scheme similar to the *mark-during-sweep* scheme proposed in [Queinnec 89]. On each node one common sweeper takes care of the sequential traversal of the node-local heap.

The scheme is based on marking with the current garbage collection cycle number, instead of marking objects black, gray, or white. The gray information is kept aside already, as this information is used by the faulting mechanism also. During the mark-phase the current cycle number represent the color black, and the previous cycle, the color white. Objects marked with lower numbers are identified garbage, waiting for the sweeper to pass by and reclaim them. At the end of the mark-phase the previous cycle number becomes a garbage indicator, just like white indicates garbage when the mark-phase is finished. When the next garbage collection cycle is started, the cycle number is incremented, effectively turning all objects to be considered white until they are marked again. New objects are always born with the current cycle number, i.e., black.

The sweeper is interleaved with the allocation routines, i.e., each time a new allocation request is received, the allocator tries to reclaim the same amount of storage from the heap. It does so starting from its current sweep position in the heap and moves forwards (viewing the heap as a circular list of objects) until the requested amount is reclaimed or no more objects marked with old cycle numbers exist.

9 Conclusion

The Emerald garbage collection scheme has reached its goals by running two mark-and-sweep collectors on each node in the distributed system:

- The global collectors cooperate to achieve a *comprehensive collection* of all garbage.
- The global collectors are *robust to node failures*. The collection progresses on the available nodes as fast as possible before they wait for the still needed, but failed, nodes to become available again. All nodes need not be available concurrently.
- The local collection ensures that local garbage is collected on a per node basis independent of the current status of other nodes, thus achieving an *expedient collection*.
- The garbage collection *faulting* mechanism has made concurrency with mutators possible and thus limit the length of pauses introduced by garbage collection on user processes.

The measurements and experiments with the implementation will present a definite evaluation of the presented collection scheme. Due to a very untimely disk crash whereby parts of the current implementation were lost, an evaluation of the prototype is, unfortunately, not available as this article goes to press.

Acknowledgement

We gladly acknowledge the comments and proof-reading by Birger Andersen. Also the comments from the anonymous referees helped clarifying several points in the presentation; they are hereby acknowledged.

References

[America 90] Pierre America, editor. Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[Appel 88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 23(7), pages 11–20, ACM, SIGPLAN, Association for Computing Machinery, Georgia, USA, July 1988.

[Appel 91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV Proceedings in: *SIGPLAN Notices* 26(4), pages 96–107, ACM SIGARCH/SIGOPS/SIGPLAN and IEEE Computer Society, TC MM / TC VLSI / TC OS, ACM Press, Santa Clara, California, USA, April 1991. Simultaneous published as

SIGARCH *Computer Architecture News* 19(2) and SIGOPS *Operating Systems Review* 25, special issue.

[ASPLOS 91] ACM SIGARCH/SIGOPS/SIGPLAN and IEEE Computer Society TC MM / TC VLSI / TC OS. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-IV Proceedings in: *SIGPLAN Notices* 26(4), ACM Press, Santa Clara, California, USA, April 1991. Also published as SIGARCH *Computer Architecture News*, 19(2) and SIGOPS *Operating Systems Review*, 25, special issue.

[Augusteijn 87] Lex Augusteijn. Garbage collection in a distributed environment. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, pages 75–93, ESPRIT, Eindhoven, The Netherlands, Springer-Verlag, June 1987.

[Bartlett 88] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* WRL Research Report 88/2, Digital, Western Research Laboratory, Palo Alto, CA, USA, February 1988.

[Beemster 90] Marcel Beemster. Back-end aspects of a portable POOL-X implementation. In Pierre America, editor, Parallel Database Systems (PRISMA Workshop) Proceedings published in: *Lecture Notes in Computer Science* 503, pages 193–228, PRISMA project, supported by the Dutch *Stimuleringsprojectteam Informaticaonderzoek (SPIN)*, Springer-Verlag, Noordwijk, The Netherlands, September 1990.

[Bekkers 92] Yves Bekkers and Jacques Cohen, editors. Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Black 86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the Emerald system. In OOPSLA'86, ACM Conference on Object-Oriented Programming, Systems, Languages and Applications, Proceedings published in: *SIGPLAN Notices* 21(11), pages 78–86, October 1986.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[Boehm 88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice & Experience*, 18(9):807–820, September 1988.

[de Bakker 87] J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors. PARLE'87, Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Proceedings published in: *Lecture Notes in Computer Science* 259, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1987.

[Gjessing 88] S. Gjessing and K. Nygaard, editors. ECOOP'88, European Conference on Object-Oriented Programming, Proceedings published in: *Lecture Notes in Computer Science* 322, Springer-Verlag, Oslo, Norway, August 1988.

[Hutchinson 87a] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, January 1987. Technical Report 87-01-01.

[Hutchinson 87b] Norman C. Hutchinson, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. *The Emerald Programming Language Report.* Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, Washington, October 1987. Also available as DIKU Report (Blue series) no. 87/22, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark and as TR no. 87-29, Department of Computer Science, University of Arizona, Tucson, Arizona.

[Jul 87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 105–106, Association for Computing Machinery, December 1987. Extended abstract only; full paper published as [Jul 88b].

[Jul 88a] Eric Jul. *Object Mobility in a Distributed Object-Oriented System.* PhD thesis, Department of Computer Science, University of Washington, Seattle, Washington, December 1988. Technical Report no. 88-12-6. Also available as DIKU Report (Blue series) no. 89/1 from Department of Computer Science, University of Copenhagen, Denmark.

[Jul 88b] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobilty in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[Juul 92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Yves Bekkers and Jacques Cohen, editors, Memory Management, International Workshop IWMM 92, Proceedings published in: *Lecture Notes in Computer Science* 637, pages 103–115, INRIA, IRISA, and ACM Sigplan, Springer Verlag, St. Malo, France, September 1992.

[Juul 93] Niels Chrisitian Juul. *Comprehensive, Concurrent, and Robust Garbage Collection in the Distributed, Object-Based System, Emerald.* PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Denmark, 1993. Available as Technical Report DIKU 93/1.

[Lang 92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Proceedings of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'92)*, ACM SIGPLAN and ACM SIGACT, Association for Computing Machinery, Albuquerque, New Mexico, USA, January 1992.

[Lieberman 83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Liskov 86] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th annual ACM Symposium on Principles of Distributed Computing (PODC'5)*, pages 29–39, Association for Computing Machinery, Vancouver (Canada), August 1986.

[Mancini 91] Luigi V. Mancini, Vittoria Rotella, and Simonetta Venosa. Copying garbage collection for distributed object stores. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

[Odijk 89] E. Odijk, M. Rem, and J.-C. Syre, editors. PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1989.

[Queinnec 89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING sweep rather than mark THEN sweep. In E. Odijk, M. Rem, and J.-C. Syre, editors, PARLE'89, Parallel Architectures and Languages Europe, Volume I: Parallel Architectures, Proceedings published in: *Lecture Notes in Computer Science* 365, pages 224–237, ESPRIT, Springer-Verlag, Eindhoven, The Netherlands, June 1989.

[Raj 91] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software – Practice & Experience*, 21(1):91–118, January 1991.

[Rudalics 86] Martin Rudalics. Distributed copying garbage collection. In William L. Schelis and John H. Williams, editors, *1986 ACM Symposium on LISP and Functional Programming, Proceedings of*, pages 364–372, ACM SIGPLAN / SIGACT / SIGART, Association for Computing Machinery, Cambridge, Massachusetts, USA, August 1986.

[Schelvis 88] Marcel Schelvis and Eddy Bledoeg. The implementation of Distributed Smalltalk. In S. Gjessing and K. Nygaard, editors, ECOOP'88, European Conference on Object-Oriented Programming, Proceedings published in: *Lecture Notes in Computer Science* 322, pages 212–232, Springer-Verlag, Oslo, Norway, August 1988.

[Scherlis 86] William L. Scherlis and John H. Williams, editors. *1984 ACM Symposium on LISP and Functional Programming, Conference Record*, ACM, SIGPLAN/SIGACT/SIGART, Association for Computing Machinery, Cambridge, Massachusetts, USA, August 1986.

[Shapiro 90] Marc Shapiro, David Plainfossé, and Olivier Gruber. *A garbage detection protocol for a realistic distributed object-support system.* Rapport de Recherche INRIA 1320, INRIA-Rocquencourt, Paris, France, November 1990.

[Shapiro 91] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, IEEE Computer Society, TC Distributed Processing, Pisa, Italy, September 1991.

[SIGPLAN 88] ACM, SIGPLAN. ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Proceedings in: *SIGPLAN Notices* 23(7), ACM Press, Georgia, USA, July 1988.

[Steele Jr. 84] Guy L. Steele Jr., editor. *1984 ACM Symposium on LISP and Functional Programming, Conference Record*, ACM, SIGPLAN/SIGACT/SIGART, Association for Computing Machinery, Austin, Texas, USA, August 1984.