Garbage Collection Curriculum

Distributed Systems Group
http://www.gsd.inesc-id.pt

Paulo Ferreira
Luís Veiga

Contents

- Introduction
- Motivation
- Terminology
- Basic Uniprocessor GC Algorithms
  - reference counting
    - deferred reference counting
  - tracing
    - mark and sweep
    - copy
  - functioning modes
    - incremental tracing
      - read barrier
      - write barrier
    - partitioned tracing
      - hybrid collectors
      - generational collectors
      - train algorithm
      - ulterior reference counting
    - system requirements
- Distributed GC Algorithms
  - reference counting
    - races
    - weighted reference counting
    - indirect reference counting
    - reference listing
  - tracing
    - concurrent distributed tracing
    - mark-and-sweep with timestamps
    - logically centralized tracing
    - group tracing
  - cycles of garbage
    - object migration
    - trial deletion
    - distributed back-tracing
    - monitoring mutator events
    - distributed train
    - group merging
    - mark propagation with optimstic back-tracing
    - DGC-consistent-cuts
    - algebra-based detection
- GC in transactional systems
  - transactional reference listing
  - transactional mark and sweep
  - atomic copy
  - replicated copy
  - GC-cuts
- Distributed GC in replicated memory systems
  - GC and DSM consistency
  - DGC on replicated memory
  - DGC for wide area replicated memory
  - Complete and scalable DGC
- Concluding Remarks
  - performance issues
  - tracing and reference-counting unification
  - final remarks
- References

Introduction

- We describe the GC algorithms characterizing them according to the following three aspects:
  - type of Algorithm, Functioning Mode and System Requirements
- Type of Algorithm
  - reference counting,
  - tracing
- Functioning Mode
  - characterizes a GC algorithm in terms of the memory on which the collector works on each run
  - e.g. a subset of all the memory and the moments in which the collection takes place with respect to applications
- System Requirements
  - a GC algorithm needs in terms of knowledge of references location, and
  - the possibility of moving reachable objects to different addresses in order to compact memory

Motivation

- It is widely recognized that manual memory management (explicit allocation and freeing of memory by the programmer) is extremely error-prone leading to:
  - memory leaks, and
  - dangling references
- Memory leaks consist on data that is unreachable to applications but still occupies memory, because its memory was not properly released:
  - memory leaks in servers and desktop computers are known to cause serious performance degradation
  - in addition, memory exhaustion arises if applications run for a reasonable amount of time
  - note that in distributed systems, memory leaks in one computer may occur due to object references present in other computers
  - furthermore, cyclic garbage complicates memory management even more, specially in distributed and/or persistent systems
- Dangling references are references to data whose memory has already been (erroneously) freed:
  - later, if an application tries to access such data, following the reference to it, it fails (i.e. referential integrity is not ensured)
  - such failure occurs because the data no longer exists or, even worse, the application accesses other data (that has replaced the one erroneously deleted) without knowing
- Dangling references are well known to occur in centralized applications when manual memory management is used:
  - such errors are even more common in a distributed environment
  - such errors harder to detect in distributed systems supporting replicated and/or persistent objects possibly with transactional semantics
- In conclusion:
  - manual memory management leads not only to applications performance degradation and fatal errors but also to reduced programmer productivity
  - thus, garbage collection, both centralized and distributed, is vital for programming productivity and systems reliability

Terminology (1)

- An object is a contiguous set of bytes
- A reference identifies an object
  - in its simplest form it is a pointer
  - we will use equivalently the term reference and pointer
- The heap designates a portion of virtual memory possibly covering the whole memory where objects are allocated
  - hereafter we will use the terms heap and memory equivalently
- We follow the standard vocabulary of the GC literature:
  - the GC-root is the set of references that forms the starting point for the GC graph tracing
  - the mutator is the application program that dynamically modifies the pointer graph (it creates objects dereferences pointers and assigns pointers)
  - the collector is the system component that identifies and reclaims unreachable objects
- In a distributed system:
  - a collector is composed of a number of threads executing in different processes we still call each one a collector
  - the mutator is actually composed of multiple independent threads running in different processes, by extension we call each of these threads a mutator

Terminology (2)

- A pointer assignment can result in:
  1. creation of a new reference to an object just created
  2. duplication of an already existing reference
  3. discarding a reference to an object
- Note that cases 2 and 3 may happen as the result of a single assignment operation
- As a side effect of pointer assignment some reachable objects become unreachable
- Unreachable objects (said to be garbage) are not needed and can be reclaimed
- A GC algorithm:
  - is safe when it does not reclaim reachable objects
  - is live if it eventually reclaims some garbage
  - is complete if it eventually reclaims all garbage
- Obviously every GC algorithm aims at being safe and complete
- However as will become clear this goal is not easy to achieve as scalability is at odds with completeness

Basic Uniprocessor GC Algorithms (BUGCA)

Basic Uniprocessor GC Algorithms (BUGCA)

- There are two fundamental uniprocessor GC algorithms
  (also known as local garbage collection – LGC – algorithms):
  - reference counting
  - tracing
- We present:
  - their most relevant advantages and disadvantages which are related to completeness, performance and memory fragmentation
  - several variants of the two fundamental algorithms which are based on the functioning modes and system requirements

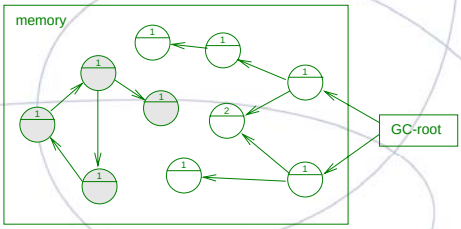
Basic Uniprocessor GC Algorithms (BUGCA)

Reference Counting

- reference counting
- deferred reference counting

BUGCA - Reference Counting (1)

- The basic idea of the reference counting algorithm is the following [Christopher84,Cohen81]:
  - a counter is associated to each object denoting the number of references to it
  - when an object is created, a single reference points to it and its counter is initially one
  - each time a reference is duplicated the object's counter is incremented
  - each time a reference to an object is discarded its counter is decremented



- Therefore, reference counting preserves the following invariant:
  - the value of an object's counter is always equal to the number of references to it

BUGCA - Reference Counting (2)

- When a counter drops to zero:
  - the corresponding object is no longer reachable
  - thus, it can be safely reclaimed
- Garbage objects are therefore reclaimed immediately after becoming unreachable
- However the converse is not true i.e.:
  - an object may be unreachable but its counter is greater than zero
  - this happens when an object is part of a cycle of unreachable objects
- When an object is reclaimed (its counter has become zero) its pointers are discarded:
  - thus, reclaiming one object may lead to the transitive decrementing of reference counts and reclaiming many other objects

BUGCA - Reference Counting (3)

- This algorithm is simple to implement, however it has three main problems:
  - it is inefficient as its cost is proportional to the amount of work done by the mutator because counters must be updated whenever references are assigned
    - this problem is addressed by variants of this algorithm such as deferred reference counting [Deutsch76]
  - the free space recovered from reclaimed objects is interspersed with reachable objects thus reducing locality of reference
    - as new objects are allocated in the free space recovered from reclaimed objects unrelated objects are interleaved in memory
    - this may lead to a situation in which the working set of an application is scattered across many virtual memory pages so that those pages are frequently swapped in and out of main memory
  - cycles are not reclaimed thus this algorithm is not complete
    - this may lead to memory exhaustion

BUGCA - Deferred Reference Counting (1)

- For some objects reference-count operations are a wasteful effort [Deutsch76]
  - large number in a short period of time with zero or small net result
    - frequent end result is the same as if they had never been performed
  - delaying these operations would greatly improve performance
    - it may also make them un-necessary to perform
  - examples include reference count increments for objects passed as function parameters and later decrements when the function returns.
    - experiments show that it eliminates 90% of reference count increment/decrement operations [Baden83].
- Reference counts may become temporarily inaccurate
  - references from local variables are not included in this book-keeping most of the time
    - references from stack are not being monitored by reference counting
  - thus, objects with zero reference count may be not all garbage
    - objects with zero count are maintained in a *zero count table (ZCT)* to prevent premature collection

BUGCA - Deferred Reference Counting (2)

- Garbage collection
  - GC may only be performed when references from stack are also accounted for
    - this stabilization is performed less frequently than reference-count operations
  - before collection, roots are traced and every element in the ZCT reachable form them is removed from the ZCT
    - the remaining objects in ZCT may be reclaimed
    - these objects are unreachable from other heap objects, from stack and from roots, therefore garbage
- Can be regarded as subtle hybrid algorithm
  - reference count among heap objects
  - tracing from roots
- Improves GC performance
  - greatly reduces reference counting overhead
    - avoids performing reference-count updates for most short-lived pointer variables from the stack
  - additional optimization performs reference count all increments before any decrements, thus avoiding the need to maintain a ZCT [Bacon01]

Basic Uniprocessor GC Algorithms (BUGCA)

Tracing

- mark and sweep
- copy

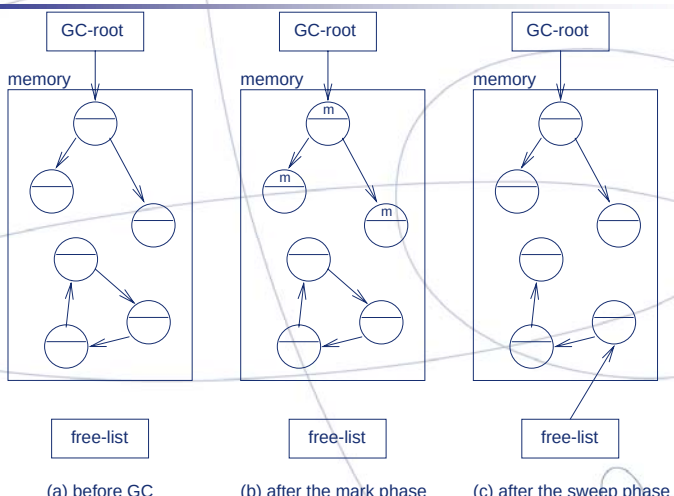
BUGCA – Tracing

- There are two algorithms of the tracing type:
  - mark and sweep
  - copy
- Tracing algorithms traverse the pointer graph from the GC-root:
  - to determine which objects are reachable
  - unreachable objects are reclaimed
- The main advantage of tracing algorithms is their ability to reclaim cycles of unreachable objects

BUGCA – Tracing – Mark and Sweep (1)

- A mark and sweep collector has two phases [McCarthy60]:
  1. trace the pointer graph starting from the GC-root marking every object found
  2. sweep i.e. examine all the heap reclaiming unmarked objects
- During the mark phase:
  - every reachable object is marked (setting a bit in the objects header for example) and scanned for pointers
  - this phase ends when there are no more reachable objects to mark
- During the sweep phase:
  - the collector detects which objects are not marked and inserts their memory space in the free-list
  - when the collector finds a marked object it un-marks it in order to make it ready for the next collection
  - this phase ends when there is no more memory to be swept
- This algorithm has two main problems:
  - fragmentation – just as with reference counting, free space recovered from reclaimed objects is interspersed with reachable objects
  - cost of the sweep phase – it is proportional to the size of the heap

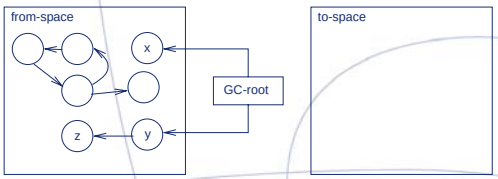
BUGCA – Tracing – Mark and Sweep (2)



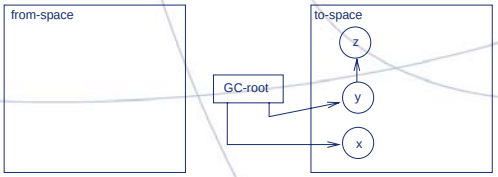
BUGCA – Tracing – Copy (1)

- The collector traces the pointer graph from the GC-root and moves each object reached to another location [Fenichel69,Minsky63]:
  - memory is compacted therefore eliminating fragmentation, and
  - allocation of objects is done linearly
- The collector works as follows:
  - divides the heap in two disjoint semi-spaces called from-space and to-space
  - during normal mutator execution objects are allocated linearly in from-space
  - once the collection starts the collector moves reachable objects to to-space
  - unreachable objects are left in from-space
  - when every reachable object has been moved the roles of the semi-spaces is exchanged (the to-space becomes the from-space and vice-versa)
    - this transition is called flipping
    - flipping is atomic w.r.t. the mutator
- An inconvenient of the copy algorithm is that only half of the memory space available is used at any point in time:
  - the to-space is a wasted resource between collections

BUGCA – Tracing – Copy (2)



(a) before collection



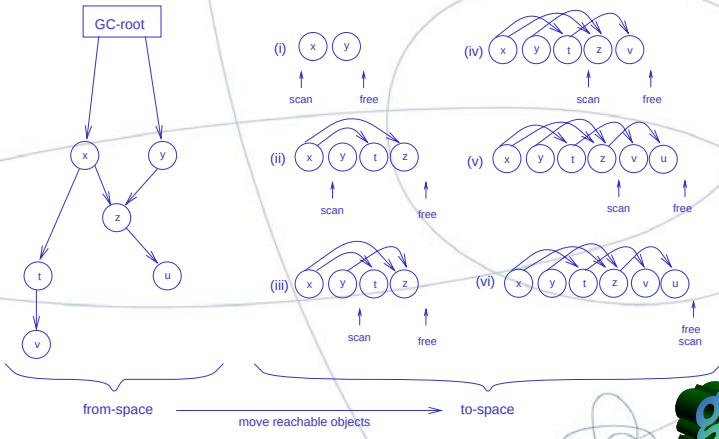
(b) after collection

BUGCA – Tracing – Copy (3)

- The Cheney algorithm [Cheney70] is a well known technique to move reachable objects from from-space to to-space:
  - objects immediately accessible from the gc-root (x and y in the next slide) are the first to be moved to to-space
  - these objects form the initial set for a breadth first traversal that is implemented with the help of two pointers:
    - free - which points to the first free address in to-space
    - scan - which points to the first object in to-space not yet scanned
  - then, the object pointed by the scan pointer (x in this case) is scanned for pointers into from-space
  - each object reached (t and z) is moved to to-space and the free pointer updated
  - in addition, the pointers in the scanned object are patched according to the new locations of the moved objects and a forwarding pointer pointing to to-space is left in their old location
  - when y is scanned, z will be reached and given the forwarding pointer there left z is not moved again (the pointer in y is simply patched)
  - this algorithm ends when every object in to-space has been scanned

BUGCA – Tracing – Copy (4)

- Cheney algorithm:



Basic Uniprocessor GC Algorithms (BUGCA)

Functioning Modes

- incremental tracing
  - read barrier
  - write barrier
- partitioned tracing
  - hybrid collectors
  - generational collectors
  - train algorithm
  - ulterior reference counting
- system requirements

BUGCA - Functioning Modes (1)

- Various functioning modes can be applied to the fundamental algorithms previously described
- The functioning modes are essentially orthogonal to these algorithms
- These modes characterize a GC algorithm in terms of:
  - the memory on which the collector works on each run (e.g. a subset of the all memory), and
  - the moments in which the collection takes place w.r.t. mutators
- The following modes are considered next:
  - incremental
  - partitioned
  - gc-only
  - concurrent

BUGCA - Functioning Modes (2)

- Incremental:
  - small units of gc are interleaved with small units of mutator execution
  - each gc pause time is smaller than in the GC-only mode
- Partitioned :
  - only a subset of the whole memory (called a partition) is garbage collected independently from the rest of the memory
  - the existence of multiple partitions is useful in systems where, for example, it is necessary to have a garbage collected heap coexisting with a manually managed heap [Cardelli89, Ellis93]
  - partitioned gc is also useful in large persistent and or distributed systems because different partitions may be collected in parallel and independently from each other
- GC-only:
  - the mutator is halted while the collector runs
  - the time interval during which a mutator is halted due to GC is called GC pause time
  - this time may be unacceptable in some cases (for example, in interactive applications where the user would be annoyed by such pauses)
- Concurrent:
  - the mutator and the collector run concurrently
  - the usefulness of this mode is that collection adds no pauses on top of time-slicing
  - in the rest of these slides when the difference is not relevant we will use the term incremental to designate both incremental and concurrent functioning modes
- Note that reference counting algorithms are inherently incremental because the collector and mutator execution is interleaved

BUGCA - Functioning Modes - Incremental Tracing

- The main issue of incremental tracing is:
  - how to ensure the correct execution of the collector when it competes with the mutator for the same data
- Concerning this issue there is no significant difference between mark and sweep and copy algorithms
- Before going into more details concerning this fundamental aspect it is useful to see:
  - how both mark and sweep and copy incremental collectors can be described as variants of an abstract tricolor marking algorithm

BUGCA - Functioning Modes - Incremental Tracing - Tricolor Marking Algorithm (1)

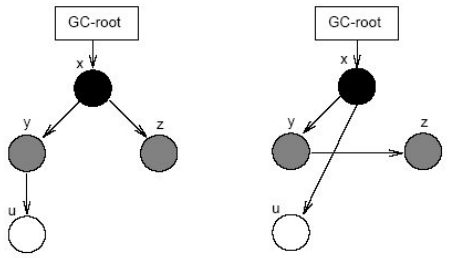
- The tricolor marking algorithm works as follows [Dijsktra78]:
  - objects subjected to collection are colored white and when the collection is finished reachable objects must be colored black
  - a collection is finished when there are no more reachable objects to blacken
    - in a mark and sweep collector this coloring can be implemented by setting mark bits objects whose bit is set are black
    - in a copy collector the coloring is the process of moving reachable objects from from-space to to-space
      - objects in the from-space are white
      - objects in the to-space are black
- The main difficulty with incremental tracing GC is that:
  - while the collector is tracing the pointer graph, as a result of mutator activity, the graph may change while the collector isn't looking !
  - if this happens the collector may not find some reachable objects

BUGCA - Functioning Modes - Incremental Tracing - Tricolor Marking Algorithm (2)

- The mutator cannot be allowed to change the pointers graph behind the collectors back in such a way that the collector fails to find all reachable objects
- Thus, there is a third color gray:
  - a gray object is one that has been reached by the collector tracing but its descendents may not have been
- Once a gray object has been scanned it becomes black and its descendents are colored gray:
  - in a copy collector the gray objects are those that have already been moved to to-space but have not yet been scanned
- Black objects may not point to white objects:
  - this invariant allows the collector to assume that it is "finished with" black objects and can continue to traverse gray objects
  - if the mutator creates a pointer from a black object to a white one it must somehow notify the collector that its assumption has been violated
  - therefore the collector must be capable of keeping track of graph changes resulting from mutator activity and retrace parts of the graph adequately
  - this ensures that the collector is aware of every change concerning the pointers graph

BUGCA - Functioning Modes - Incremental Tracing - Tricolor Marking Algorithm (3)

- Suppose that object x has been completely scanned and therefore blackened its descendents:
  - y and z have been reached and grayed
- Now suppose that the mutator swaps the pointer from x to z with the pointer from y to u:
  - the only pointer to u is now in object x which has already been scanned by the collector
  - this violates the invariant black object x pointing to white object u
  - if the tracing continues without any coordination:
    - y will be blackened
    - z will be reached again from y, and
    - u will never be reached at all, and hence will be unsafely considered garbage
- This problem is solved by coordinating the mutator with the collector:
  - this can be done with the following two techniques
    - read barrier
    - write barrier



(a) initial graph

(b) graph modified concurrently by the mutator

BUGCA - Functioning Modes - Incremental Tracing – Read Barrier

- A read barrier is used to detect when the mutator attempts to read a pointer to a white object
- Immediately the collector scans the object and colors it gray
  - since the mutator cannot read pointers to white objects it cannot write them into black objects
- There are many incremental collectors using a read barrier [Appel88,Baker78,Brooks84, Huelsbergen93, Queinnec89, Zorn89]
- We describe here a representative example using a copy algorithm
- The best known incremental copy collector is Bakers [Baker78]
  - a collection starts with a flip that conceptually invalidates for mutator accesses all objects in from-space and moves to to-space all objects directly reachable from the gc-root
  - then the mutator is allowed to resume
  - any object in from-space that is accessed by the mutator must first be moved to to-space
    - this is enforced by the read barrier
  - thus the mutator is never allowed to see pointers into from-space; if so, the referent is immediately moved to to-space
  - in terms of the tricolor marking algorithm:
    - objects in to-space that were already scanned are black objects
    - objects in to-space but not yet scanned are gray
    - objects still in from-space are white
    - new objects created by the mutator while the collection is taking place are allocated in the to-space and are colored black

BUGCA - Functioning Modes - Incremental Tracing – Write Barrier

- A write barrier detects when the mutator tries to write a pointer into a black object
  - when that happens the write is trapped or recorded [Appel88, Bohem91, Demers90, Dijsktra78, Steele75, Yuasa90]
- We describe an interesting copy collector using a write barrier; it is called replication-based GC and works as follows [Nettles92, O'Toole93]:
  - while the collector moves objects to to-space the mutator continues to access the from-space versions of objects rather than the replicas in to-space
  - when every reachable object has been moved to to-space a flip is performed and the mutator then starts seeing the to-space replicas
  - this technique eliminates the need for a read barrier because all the reachable objects are conceptually moved to to-space at once when the flip occurs
  - however it is necessary to have a write barrier because the mutator sees the old version of objects in from-space
  - if an object has already been moved to to-space and then the mutator writes into it, the replica in to-space is now out of date w.r.t. the version seen by the mutator
  - thus the write barrier catches all mutator writes and stores them in a log in order to allow the collector to propagate those modifications to the to-space replicas when the flip takes place
  - in other words, all the modifications to objects in from-space must be applied to the corresponding replicas in to-space so that the mutator sees the correct values after the flip

BUGCA - Functioning Modes - Partitioned Tracing

- The application of the partitioned mode to the basic GC algorithms reference counting and tracing results in the following two variants:
  - hybrid collectors
    - each partition is traced independently from the others and cross-partition references are handled with a reference counting algorithm [Bishop77,Moss89,Mahe97a]
    - this variant is very interesting for large distributed and for persistent systems
  - generational collectors
    - aims at reducing the GC pause time by decreasing the amount of memory that has to be collected

BUGCA - Functioning Modes - Partitioned Tracing - Generational Collectors

- It takes advantage of the following empirical observation:
  - in many applications most objects are reachable for a very short length of time only a small portion remains reachable much longer [Hayes91,Lieberman83,Moon84,Ungar87]
- Thus the idea is to separate the objects in at least two groups and collect the first group more often:
  - those objects that are reachable only for a short length of time belong to the first group
  - the others belong to the second group
- Both mark and sweep and copy algorithms can be made generational:
  - we focus on the copy algorithm

BUGCA - Functioning Modes - Partitioned Tracing - Generational Copy Collector

- Objects are segregated into multiple partitions by age
  - each partition is called a generation
  - younger generations are collected more often than older generations
  - the age of an object is approximated by the number of collections it has survived
- To avoid the cost of successive scans and moves from from-space to to-space of those objects that remain reachable for a long time:
  - partitions containing older objects are collected less often than the younger ones
  - thus, once objects have survived a certain number of collections they are moved to a less frequently collected partition instead of to-space
- To allow young generations to be collected without having to collect the older ones the collector must be capable of finding pointers into the young generations
  - this requires either the use of a write barrier similar to the one we found in the incremental functioning to keep track of such cross-partition pointers [Appel89a,Diwan92, Moon84, Ungar88,Wilson92], or
  - indirect pointers from older to younger generations [Lieberman83]
- The set of references pointing from older to younger generations is usually called remembered-set
  - when a younger generation is collected the pointers in the corresponding remembered-set are part of the GC-root

BUGCA - Functioning Modes - Partitioned Tracing - Incremental Generational Collector (Train Algorithm) (1)

- Aims at reducing maximum GC pause times [Hudson92]
- Applies to older generation (mature objects that survived collection in the younger generation - nursery)
- Mature object space is divided in *cars* of fixed memory size
  - in Persistent MOS [Moss96], cars are pages of the database.
- GC collects at most one *car* each time it runs
  - GC is incremental
  - imposes bounds on maximum pause time
    - with a high degree of confidence
      - » car size is determined by expected/desired pause time
    - *absolute* assurance imposes penalties on mutators
- Main idea: move objects closer and closer to the cars and trains they are referenced from
  - cluster related objects (referenced and referring) in the same car or, at least, in the same train
  - when there are no more inter-car or inter-train references, the whole car or train can be collected
  - can be regarded as a variation of object migration to minimize inter-car references (and garbage)
  - intra-car garbage (cyclic or acyclic) is collected by tracing
- Cars are grouped in trains with variable number of cars
  - cars are totally ordered
    - trains are ordered and cars are ordered (within a train)
  - objects surviving nursery collection are inserted in cars
    - in any train that is not being collected
      - generally the first train in the last car
      - second train if the first is already being collected
      - new train if train size limit is reached

BUGCA - Functioning Modes - Partitioned Tracing - Incremental Generational Collector (Train Algorithm) (2)

- Train collection
  - always performed on the first train
  - pointers to the first train are examined
    - sources include roots or objects in other trains
    - if there are none, the whole train is garbage and is reclaimed
    - otherwise, first car of first train is examined
  - trains record references from objects in higher trains
    - train remembered-set
- Car collection
  - pointers to the first car are examined
    - cars record references from objects in higher cars (including higher trains)
      - car remembered-set
  - if there are none, the whole car is garbage and is reclaimed
    - otherwise, move objects to the highest car they are referenced from
  - inter-car object movement
    - if not enough space in car, move to higher car or create new one if none available
      - car ordering forces object to be moved to the higher train it is referenced from
    - if there are pointers from objects in the nursery, the object is moved to a later train
  - when there are no more pointers to first car
    - the whole car is garbage and may be reclaimed
  - when there are no more pointers to any of the cars in the train
    - the whole train is garbage and all its cars may be reclaimed

BUGCA - Functioning Modes - Partitioned Tracing - Ulterior Reference Counting (1)

- Addresses Tracing vs. Reference Counting trade-off [Blackburn03]
  - tracing algorithms
    - high throughput
      - generational approach increases performance
    - higher maximum pause times due to full-heap collections
    - complete w.r.t. cycles
  - reference counting
    - lower throughput
      - costly pointer operations hurt mutator performance
      - partially addressed with Deferred Reference Counting [Deutsch76]
    - higher responsiveness
      - no pauses due to full heap collections
    - unable to detect cycles without additional detection mechanism

BUGCA - Functioning Modes - Partitioned Tracing - Ulterior Reference Counting (2)

- Combine both approaches where they suit best (*Generational-RC hybrid*)
  - generational tracing for newly allocated objects (*nursery*)
    - pointer mutations are ignored
    - objects surviving one collection are copied to mature space
      - reference-counting of survivors is piggy-backed on the tracing and copying
  - reference counting for mature objects (*mature space)*
    - deferred reference counting
      - Generalized to heap objects (adding to registers and stack variables)
    - variant of trial deletion to detect cycles in the mature space
      - ensures completeness
  - integration of two spaces and two algorithms
    - nursery objects referenced by mature objects
      - targeted via remembered set included in tracing roots
    - mature objects references by nursery objects
      - possible high number of mutations
        - » consequent reference-count increments/decrements) are avoided deferred reference counting
        - » further, it avoids reference-count increments/decrements “discarding” mutations performed on short-lived objects

BUGCA - System Requirements (1)

- Tracing garbage collectors traverse the pointer graph
  - for this purpose the collector must be capable of finding the pointers inside the GC-root and inside each reachable object
- In programming languages providing runtime type information it is possible to differentiate pointers from raw data with certainty:
  - this is the case of lisp and smalltalk where there is enough type information that can be used to determine object layouts including the locations of embedded pointers [Appel89,Goldberg89,Steenkiste87]
- However there are cases in which such runtime type information is not available:
  - as is the case with the programming languages C and C++
  - in such environments a possible solution consists of:
    - either a preprocessor [Edelson92], or
    - the compiler [Bohem91a, Ferreira91, Samples92] to statically generate type information for each data type
  - another alternative is to take advantage of some specific language features:
    - e.g. smartpointers in C++ [Detlefs92,Edelson92a] to generate the pointers locations

BUGCA - System Requirements (2)

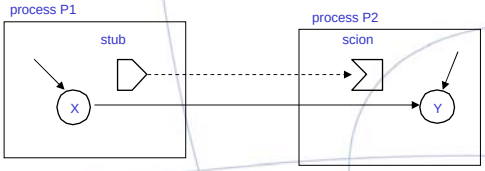
- Another solution, when runtime type information is not available, is called conservative GC [Bartlet88, Bohem93]:
  - it relies on a conservative pointer finding approach i.e. the collector treats anything that might be a pointer as a pointer
  - thus, any properly aligned bit pattern that could be the address of an object is actually considered to be a pointer to that object
- There are a few problems concerning this approach:
  - first, the conservative interpretation of ambiguous data
    - e.g. considering an integer as a pointer may lead to consider garbage objects as being reachable
    - this waste of memory can be a serious problem for memory intensive applications [Russo91, Wentworth90]
  - second, given that the collector does not differentiate raw data from pointers it has to scan all the data inside reachable objects
    - this extra cost of scanning increases the GC pause time
  - finally, a copy collector cannot move reachable objects and patch the corresponding pointers because a non-pointer might be considered to be a pointer and would be mistakenly patched
- However, in spite of these problems:
  - there are cases for which a conservative approach is adequate, and
  - sometimes the only that is feasible [Weiser89]

Distributed GC Algorithms (DGCA)

Distributed GC Algorithms (DGCA)

- In this section we present the most interesting GC algorithms found in the literature for RPC based distributed systems (i.e. with no support for persistence, distributed shared memory, transactions, etc.)
- These algorithms are extensions of the basic reference counting and tracing:
  - reference counting algorithms
    - more scalable
    - mostly incomplete, i.e., unable to detect and reclaim distributed cycles of garbage (unless special techniques are used)
  - tracing algorithms
    - inherently complete, i.e., able to detect and reclaim distributed cycles of garbage
    - non-scalable
      - require extra synchronization among processes
      - may need distributed consensus
  - hybrid approaches (proposed initially by [Dickman91,Juul92]
    - combine aspects from both families trying to achieve best of both worlds, i.e. completeness and scalability

DGCA- Data Structures



- There are certain data structures common to many DGCAs:
  - GC-stubs and GC-scions
- A stub describes an outgoing inter-process reference:
  - from a source process to a target process (e.g. from object X in P1 to object Y in P2).
- A scion describes an incoming inter-process reference:
  - from a source process to a target process (e.g. to object Y in P2 from object X in P1).
- Stubs and scions may not impose any indirection on the native reference mechanism
  - thus, in such systems, they do not interfere either with the structure of references or the invocation mechanism.
  - they are simply GC specific auxiliary data structures.
  - thus, stubs and scions should not be confused with (virtual machine) native stubs and scions (or skeletons) used for remote method invocations

Distributed GC Algorithms (DGCA)

Distributed Reference Counting

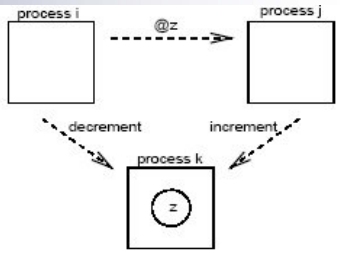
- races
- weighted reference counting
- indirect reference counting
- reference listing

DGCA - Reference Counting

- Each process in the system holds one partition that is collected independently from the rest of the memory with a tracing algorithm
  - cross-partition references are managed with a reference counting algorithm
- The extension of the uniprocessor reference counting algorithm to handle cross partition references poses some problems:
  - these problems generally called race conditions arise because objects counters have to be incremented and decremented through messages exchanged between processes
  - such messages must be delivered reliably and in causal order [Birman91, Lamport78] to ensure safety and liveness:
    - increment and decrement messages are not idempotent and must not be duplicated or lost
- There are two types of race conditions both possibly leading to the unsafe reclamation of a reachable object:
  - decrement/increment
  - increment/decrement

DGCA - Reference Counting – decrement/increment race

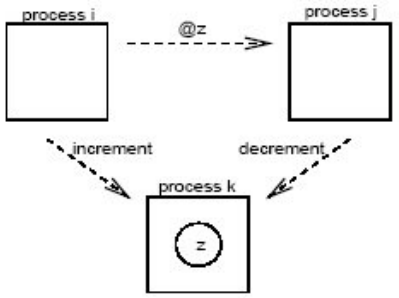
- Suppose that process i holds a reference to object z in process k and sends a message to process j containing @z (a reference to z)
- Process j receives this message and sends an increment message to k concerning object z
- Concurrently, process i deletes its reference to z and sends the corresponding decrement message to k
- If the decrement message arrives first at k, then z is considered to be unreachable and is unsafely reclaimed



- At first glance it seems that this race problem could be solved by simply making the sender (process i) conservatively emit the increment message before sending @z to j
- However this does not solve the problem and leads us to the scenario for the second race problem

DGCA - Reference Counting – increment/decrement race

- The sender process issues the increment message (not the receiver as in the previous case)
- Suppose that process i holds a reference to object z in process k and sends a message to process j containing @z
- Now, j receives this message and immediately discards it
- Therefore, it sends a decrement message to k concerning z
- Concurrently, process i sends the corresponding increment message to k
- Once again if the decrement message arrives first at k, z is unsafely reclaimed

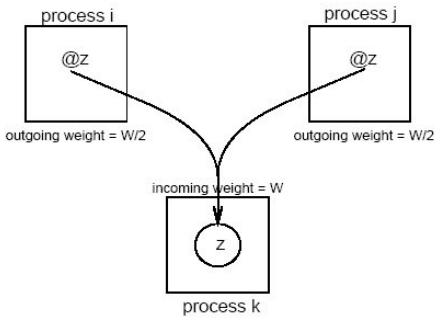


DGCA - Reference Counting – solving the race problem

- An obvious solution for these two race problems is to:
  - acknowledge the increment message before sending a decrement or a reference to a remote process respectively
  - however this introduces communication overhead and the GC algorithm is still not resilient to failures
- For these reasons there are many variants of the reference counting algorithm that can be grouped in the following main categories:
  - weighted reference counting [Bevan87,Watson87]
  - indirect reference counting [Goldberg89,Piquer91]
  - reference listing [Birrell93,Mahe94,Shapiro91,Shapiro92]
- Each one of these variants avoids the transmission of increment messages therefore solving the two race problems previously described

DGCA - Reference Counting - Weighted Reference Counting (1)

- Each cross-partition reference has two associated weights:
  - a weight at the source process (outgoing weight)
  - a weight at the target process (incoming weight)
- For any object z:
  - the incoming weight must be equal to the sum of the outgoing weights associated to the cross-partition references pointing to z



DGCA - Reference Counting - Weighted Reference Counting (2)

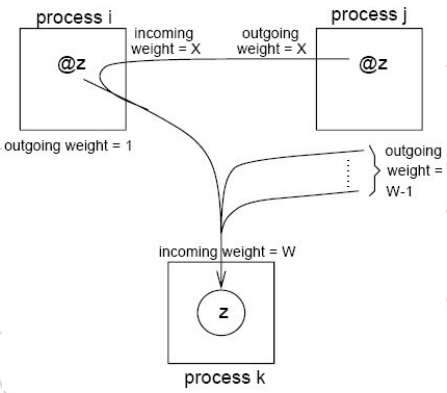
- When a cross-partition reference is first created:
  - both incoming and outgoing weights are equal (a even positive value usually a power of two)
- When the holder of a reference passes it through a message to another process:
  - it divides the current outgoing weight in two parts (normally equal)
  - retains one and sends the other along with the message
  - when the receiver process receives the message it associates to the new outgoing cross-partition reference the weight just received
  - if a cross-partition reference to the same object already exists, it simply adds the received weight to the current one
- Thus, the sum of outgoing weights is always kept equal to the incoming weight at the target process

DGCA - Reference Counting - Weighted Reference Counting (3)

- When a process deletes an outgoing cross-partition reference:
  - it sends a decrement message with the associated weight to the target process
  - the target process receives that message and subtracts the received weight from its incoming weight corresponding to the cross-partition reference
  - when the incoming weight becomes zero then the corresponding object may be reclaimed
- Note that this algorithm solves the two race problems previously described but still needs reliable communication no loss or duplication for ensuring safety and liveness
- In addition, this algorithm has the following problem:
  - the limit imposed on the number of times a reference may be sent to another process is limited by the initial associated weight
  - this problem is solved by the use of an extra indirection

DGCA - Reference Counting - Weighted Reference Counting (4)

- Suppose that:
  - process i holds an outgoing cross-partition reference pointing to z in process k, and
  - that the associated weight has dropped to 1 (i.e. no more division of the weight is allowed)
- Now, if process i needs to send @z to process j:
  - the collector creates a new incoming cross-partition reference coming from j that refers indirectly to z through the original outgoing cross-partition reference that points to k



DGCA - Reference Counting - Indirect Reference Counting (1)

- Weighted reference counting suffers from the reference weight underflow problem
  - a unitary weight cannot be further divided
  - an indirection element is created and inserted in the references
    - indirection element has full reference weight and therefore, allows further weight division and reference duplication
  - indirection element is permanent and prevents objects from later being reclaimed
    - when this happens to a large number of dependent objects (chains of references) it's called domino problem
- Solution: each reference holds a pointer to the node where it was copied from [Rudalics90,Piquer91]
  - each reference holds counter of times the reference was copied to other nodes
  - when a reference is duplicated, the counter is incremented
    - the new reference points to the existing one and holds its own duplication counter
  - sequence of duplications of the same reference through a series of nodes creates an inverted tree of indirections
    - root is the node where the reference was initially created

DGCA - Reference Counting - Indirect Reference Counting (2)

- Solves underflow problem
  - every reference has an indirection element, however it is not permanent
  - it also avoids distributed races among messages regarding reference creation and deletion present in reference counting
- Allows deletion only of the references placed at leaves of the inverted tree of reference indirection
  - preserves a lot of garbage indirections in intermediary nodes
    - references are no longer being used but are kept until all the duplicated references in child nodes have been deleted.
  - the structures preserving garbage reference indirections have an important memory overhead

DGCA - Reference Counting - Reference Listing

- For remote references, a reference counter is replaced by a list of referring nodes (reference-list) [Piquer91]
  - used only for inter-node references
    - local (intra-node) references may be managed via reference counting or tracing
  - reference-list is duplicate-free
    - multiple references from the same node share the same element in the list
  - reference duplication triggers insertion message
    - referring node is inserted in the reference list
  - reference deletion triggers removal message
    - referring node is deleted from the reference list
- Creation of inter-node references
  - Creation of scions and message to create stub counterpart
- Destruction of inter-node references
  - No specific messages to delete stubs or scions
  - Reference listing messages are idempotent
    - reference-list messages carry all live stubs w.r.t. destination process
    - stubs in message are matched with counterpart scions in process; scions without stub are deleted
    - resilient to message loss and duplication
    - distributed races are avoided
- Reference-lists occupies more storage than reference-counters

Distributed GC Algorithms (DGCA)

Tracing

- concurrent distributed tracing
- mark-and-sweep with timestamps
- logically centralized tracing
- group tracing

DGCA - Tracing

- Sequential distributed tracing [Ali84]
- Mutator must be stopped at every node
  - Stop-the-world approach
- Any node may decide to begin a collection
  - Coordinator is decided statically or dynamically
    - Dynamic selection needs distributed consensus
  - Instructs mutators to stop
- Marking phase
  - Marks are propagated along local and inter-process references via marking messages
  - Global Marking phase termination
    - Marking phase terminated at every node
    - No more marking messages in transit
- Sweeping phase
  - Executed concurrently by all nodes

DGCA – Concurrent Distributed Tracing

- First presented by [Hudak82]
  - adaptation to distributed environments of the algorithm found in [Dijsktra78]
- Functional languages (single root for all distributed graph)
- Each recursive marking step becomes an autonomous task executed concurrently
- Each nodes maintains two lists of active tasks
  - mutator tasks
  - GC marking tasks
- Task termination
  - each task terminates after terminations of all the tasks spawned from it
- Tri-color marking adapted to distributed scenario
  - white – objects unmarked yet
    - initially all objects are marked white
    - white objects constitute garbage at the end of marking
  - gray
    - marked objects but whose spawned tasks have not finished yet
  - black
    - newly created objects
    - marked objects whose spawned tasks have already finished
- Mark phase termination implies
  - distributed consensus
  - sweep phase starts only after termination of marking phase in all processes
- High number of marking tasks spawned
  - demanding w.r.t. network traffic and processing power

DGCA - Tracing - Mark-and-Sweep with Timestamps (1)

- Mark-bits are replaced with time-stamps [Hughes85]
- Local GC propagates time-stamps from scions to stubs
  - local roots are marked with current time
  - local roots and scions are traced in decreasing order
    - this guarantees that stubs are marked with maximum time-stamp of the scions and/or local roots that lead to them
    - marks associated with live objects are increased
    - marks associated with garbage objects remain constant
  - DGC propagates marks from stubs to corresponding scions in other processes
    - upon reception, scion time-stamp is updated if the value received is greater than its current time-stamp
      - this is not always the case, e.g., if used, while it is not propagated locally, this is recorded as a REDO
  - each node keeps the maximum value received in messages and already propagated locally
  - when the receiver acknowledges reception of all messages, the sender can update its REDO value

DGCA - Tracing - Mark-and-Sweep with Timestamps (2)

- A global minimum of all REDO values is calculated
  - every object with time-stamp below MinREDO is garbage and is reclaimed
- Minimum calculation depends on global termination algorithm [Rana83]
- Process clocks should be synchronized and message latency bounded
- Allows concurrent marking of all processes
  - synchronization needed only to decide new minimum value.
- Needs all processes to make progress
  - if one process does not cooperate, global minimum will freeze
  - no new garbage can be detected from that moment

DGCA - Tracing - Logically Centralized Tracing (1)

- Goal: obtain information about global reachability of each object from a high-available logically centralized service[Liskov86]
  - high availability needs replication
    - replicas exchange information via “gossip” messages [Fisher82]
- Each process performs asynchronous local garbage collection (LGC) and computes accessibility information scions (inlists) and stubs (encoded in locally known paths).
  - central service collects accessibility information from each node
  - central service maintains view of the global graph and when requested, informs a node of the scions it should delete since there are no longer any stubs pointing to it.
    - view is inconsistent but safe with conservative approach
    - to ensure safety, all messages must be time-stamped
    - process clocks must be loosely synchronized
    - messages should have bounded latency
  - does not need global synchronization of nodes
  - does not need all processes to participate continuously

DGCA - Tracing - Logically Centralized Tracing (2)

- High communication overhead from all nodes to the central service
  - congestion
    - the service performs all DGC tasks and not only those related to cycle detection
  - global collection delay
  - garbage collection of cycles may need several iterations
    - added communication and processing load
- Problems found in [Rudalics90] later addressed in [Ladin92]

DGCA - Tracing - Group Tracing

- Distributed cycle detection within groups of processes [Lang92]
  - does not require every process to participate
  - initiated only when LGC cannot reclaim enough memory
    - groups are created dynamically
    - groups may have any number of nodes
    - groups may share nodes
  - distributed tracing within group
    - references from processes outside the group are conservatively considered as GC-roots
  - node failure is handled by group creation dynamism
    - a new group is formed with the surviving processes
    - allows detection to continue
  - not scalable to whole system
    - scalable to large numbers of small groups
    - "final" group to collect remaining cycles implies tracing the whole system.
  - doubts about safety w.r.t. concurrency with the mutator raised in [Mahe97]

Distributed GC Algorithms (DGCA)

Distributed Garbage Collection Algorithms of Cycles of Garbage (DGCACY)

- object migration
- trial deletion
- distributed back-tracing
- monitoring mutator events
- distributed train
- group merging
- mark propagation with optimstic back-tracing
- DGC-consistent-cuts
- algebra-based detection

DGCACY - Types of Detectors of Garbage Cycles

- There are two types of DGCA for distributed cycles
  - all-at-once detectors
    - detect all existing cycles simultaneously
    - impose additional work globally to the system
      - even when and where there are no cycles
      - continuous extra-cost (time/space/messages) leads to wasted work
  - suspect-based detectors (or per-cycle detectors)
    - detect one cyclic graph of garbage each time
    - need one candidate object to initiate
      - suspect of belonging to a distributed garbage
      - heuristic is needed to select a candidate
      - algorithm confirms if it actually belongs to a cycle
    - several cyclic graph detections may run concurrently
    - impose specific work for each cycle detection
      - restricted to processes related with cycle being detected

DGCACY - Reference Counting – Object Migration (1)

- Proposes use of object migration to co-locate all objects belonging to a cycle [Bishop77] and reclaim them with a tracing collector
- Cycle consolidation
  - initiated with a suspected object
    - chosen based on a heuristic
      - locally unreachable
      - long time without remote invocations
      - increase in reference deletion messages received by the node, since the object was created or last invoked
  - move/migrate all objects belonging to a distributed cycle to a single process
    - every suspected object is migrated to one of its client nodes
    - client nodes must be explicitly known
      - this imposes de use of indirect reference counting or reference listing algorithm
      - distributed reference counting or weighted reference counting cannot be used
  - transform distributed cycle of garbage into a local one confined to one process
    - then, any local tracing algorithm may be used to reclaim the cycle
- Reference counting extended with cycle consolidation provide completeness

DGCACY - Reference Counting - Object Migration (2)

- Limitations:
  - may need a large number of iterations to consolidate even simple cycles
  - migration requires extensive system homogeneity among nodes
  - consolidation of large cycles may cause memory and bandwidth overload
  - heuristics are not optimal
    - live objects may be migrated along with garbage ones
    - very difficult to select the best process to receive the entire cycle
- Restrict direction of object migration to a total order among nodes to ensure all objects in a cycle converge to the same node [Shapiro90]
  - virtual object migration
    - objects are not physically transferred, just marked as belonging to a different logical space
    - tracing of logical spaces may trigger additional inter-process messages
- Using a fixed dump node to retain all objects in cycle [Gupta93]
  - usable with all variations of reference counting
    - no need to know client nodes explicitly, since objects are always migrated to the same node, the dump.
  - unscalable, single-point of failure
  - some live objects are still migrated to the dump

DGCACY - Reference Counting - Object Migration (3)

- Controlled Migration [Mahe95]
  - distance heuristic to identify likely garbage
    - distance: minimum number of inter-node references from a local root to objects
      - local roots have distance 0
      - new scions have initial distance set to 1
    - DGC messages propagate distances from stubs to corresponding scions
    - LGC propagates increased (unitary increments) distances from scions to stubs
    - minimum distances are always propagated
      - scions are traced by increasing distance order
        - » each object needs to be traced only once
    - distance over threshold T indicates object is likely garbage
      - T may be a small multiple of the expected maximum distance
        - » T too small may cause extra-migrations of live objects
        - » T too large delays cycle detection
  - object batching for migration
    - determines all the objects that should be migrated along with the suspected
    - combines distance heuristic and ordered scion tracing
      - objects referenced by, and with distance equal or larger, to other objects being migrated are very likely to be also garbage
      - other objects, also referenced by suspected objects, but with lower distance, should not be migrated

DGCACY - Reference Counting - Object Migration (4)

- node selection: select what is the best node where to migrate the entire cycle
  - node selection is heuristic
    - node hints are propagated along with distances above T
    - propagate maximum of
      - » node that the stub refers to
      - » destination hint of the scion that leads to the stub or local node ID if hint is absent
    - distance threshold $T_2 >$ T + Exp. Max. Cycle Size + Exp. Max. Rounds for hints to reach all nodes
      - » triggers object migration
    - node ordering ensures node selection convergence
      - » migration is allowed only in the direction of increasing node IDs
- comparison to other migration techniques
  - avoids single-point of failure of the fixed-node approach
  - minimizes migration to different sites
  - transfers groups of objects in batch in a single message
  - additional complexity with two tracing mechanisms
    - object distances and node ID hints

DGCACY - Reference Counting - Trial Deletion

- Proposes trial deletion of objects to detect cyclic garbage [Vestal87]
- Separate set of reference-count fields for each object
  - used to propagate the effects of (trial) simulated deletions
    - reference-counts decrements
  - starts with an object suspected of belonging to a distributed garbage cycle
- Simulates the recursive deletion of the object and all its referents
  - if the reference-counts of all the objects drop to zero, a cycle has been found
    - this ensures that there are no other references from objects outside the cycle being tentatively deleted
- Recursive freeing is an unbound process
  - size of cycle is not anticipated
  - poor candidate selection (e.g., a live object) will lead to trial deletion of a large number of objects in many processes
- Problems detecting mutually referencing distributed cycles

DGCACY – DGC with Distributed Back-tracing (1)

- Back-traces references from objects in order to find global roots referencing them [Mahe97]
- Initiated on a candidate-object
  - a suspect of belonging to a distributed cycle of garbage
- Stops when:
  - finds local roots
    - and aborts detection, or
  - all objects leading to the suspect have been back-traced
    - a cycle has been detected correctly
- Based on two mutually recursive procedures
  - one that performs local back-tracing
    - every object holding out-going references holds leader field referring objects with incoming references targeting it
  - another that performs remote back-tracing
    - iorefs (representing remote refs) must be marked with trace-id's that visited it
    - ensures termination
    - allows multiple overlapping detections

DGCACY - DGC with Distributed Back-tracing (2)

- Detections result in direct acyclic chaining of recursive remote procedure calls
- Copies of references (local and remote) are subject to transfer-barrier
  - updates iorefs (referring to both stubs and scions collectively)
  - ensures safety
  - distributed transfer barrier (for remote references) may need to send extra messages to processes
    - these messages must be guarded against delayed delivery
- Applied in the context of DGC in CORBA [Rivera97]
  - addressing detailed implementation issues
  - real environments/real systems
  - uses commercial-of-the-shelf (COTS) software

DGCACY - DGC by Monitoring Mutator Events (1)

- Performs DGC by storing and monitoring relevant mutator events[Louboutin97]
- Introduces alternative to tracing and lazy approach to DGC
  - lazy vs. eager log-keeping other algorithms use
  - does not need immediate exchange of control messages
    - representing inter-process references
    - may be postponed until they become necessary
- Analyses mutator computation instead of object graph
  - only global roots are considered
    - contain every object that is (or has been) target of remote references
  - edge-creation events (reference creation and duplication)
  - edge-destruction events (reference deletion and modification)

DGCACY - DGC by Monitoring Mutator Events (2)

- Global roots conceptually exchange log-keeping control messages
  - each global root appears/behaves as a process conceptually exchanging log-keeping messages
  - messages regarding only edge-creation and edge-destruction events
- Direct Dependency Vectors – DDV (time-stamps)
  - vector clock with one entry per each global root
  - each entry is a monotonically increasing counter
- Reflect causality of mutator events
  - value zero denotes no message has been received from corresponding global-root
  - value _N_ denotes that last message received from corresponding global-root was an edge-destruction message
  - DDV for a specific events is derived combining DDV of local predecessor event and DDV of direct remote predecessor.
  - event Vector time diff event DDV
    - it represents all events causally preceding it
    - dynamically calculated from increasingly accurate versions of DDV
    - complete transitive closure corresponds to full vector-time

DGCACY - DGC by Monitoring Mutator Events (3)

- Processes maintain and propagate causality information
  - each vertex (scion) maintains a two-dimensional log of DVV messages
    - creation/destruction of incoming edge to the vertex
      - vector $DV_i[i]$ is updated with latest event of source vertex (stub)
    - lazy delivery of DVV messages
      - multiple edge-creation and one edge-destruction can be bundled in one delivery
    - vertex objects are clustered in processes
      - entity that actually sends and receives messages
      - nevertheless, different objects use different event indexes
    - complete causal-cut including edge-creation and destruction identifies garbage
- Pros
  - resilient to message duplication and loss
  - lazy approach avoids synchronization bottleneck and races
- Cons
  - unbounded detection latency
    - w.r.t. all garbage and not just cyclic garbage
  - space overhead for storing message logs

DGCACY: Train Algorithm for Distributed Systems (1)

- Adaptation of Train algorithm (DMOS) to distributed scenario [Hudson97]
  - objects reside in cars of fixed size
  - each car resides on a single node
  - trains comprise several cars
    - possibly spanning several nodes
    - allows detection of inter-process cyclic garbage
    - each train has a fixed master node responsible for the train
  - there must be always *two* trains in each node
  - cars and trains are ordered as in the centralized train algorithm
    - nodes are also ordered , thus **node:train** is a completely ordered set
  - may be used with or without object migration
- Pointer Tracking (events monitoring creation, deletion and transfer of pointers)
  - s: inform object 's (*o*) home node (*H*) that a pointer to *o* has been sent from *A* to *B*
  - r: inform *H* that a pointer to *o* sent from *A* has been received at *B*
  - d: inform *H* that a pointer to *o* sent from *A* and received at *B* has been deleted from B's message buffers
  - *: inform *H* that a new pointer to *o* has created at *A*
  - -: inform *H* that a pointer to *o* has deleted at *A*
  - copy pointer to *o(H) from* A *to* B generates ***s*** event at *A,* and causally, events ***r, +, d*** at *B*
  - events are optimized
    - avoiding sending redundant **+** and **–** messages from *A,B* to *H*
      - send only those that may change object from unreachable to reachable (and vice-versa) in node
    - events piggy-backed on other messages, event compression in messages, and event combination on nodes

DGCACY: Train Algorithm for Distributed Systems (2)

- Car and Train Management
  - nodes (e.g., *H)* know which cars have pointers to its objects (e.g. *O(H))*
    - table indicates cars (C) that have pointers to *o(H)*
      - + and – events update table entry for pointers to *o(H)* in each car
      - this table acts as a *remembered-set*
    - a *Sticky-remembered-set is needed* for completeness
      - accumulates every car that was ever known by node *H*
      - current *remembered-set* is always a subset of the *sticky-remembered-set*
    - each train has a *master node*
      - node where train was created
      - master node creates, manages and cleans up the train
  - Nodes holding cars of a train are linked
      - logical token passing ring, each node knows its *successor.*
    - Joining: New nodes (e.g. *X*) are always inserted after master node
      - Successor(A, n:A) → Successor(Successor(X, n:A))
    - Leaving: nodes which all cars belonging to train have been reclaimed
      - node sends *leave* message (indicating its sucessor)
      - message is propagated around the ring until it reaches X's predecessor
        - » predecessor node cuts X out of ring,
        - » links to its successor and acknowledges X
        - » X must continue to forward messages until notified
      - multiple nodes trying to leave simultaneously
        - » adjacent nodes leaving at the same time update *successor* field in message
        - » safe because messages circulating the ring are never over-taken

DGCACY: Train Algorithm for Distributed Systems (3)

- Basic Garbage Collection
  - finding out that there are no pointers into a train from objects in cars outside the train
    - each node holds a *change-bit* regarding changes in sticky-*remembered-set* of each train
      - *Changed-bit* is set to *true* initially at master node
      - *Changed-bit* at a node is set to *true* when a the node joins the train
  - relaying "no-refs" *token* around the logical ring with associated value
    - token starts at master node (e.g., *A*) with value {A}, other nodes may *re-start* token
    - token is always *held* by one of the nodes in the ring
    - node (e.g., *Y*) receiving the token either *holds* or *relays* it
      - Rule 1: if *Y* has external pointers in train *sticky-remembered-set,* it must **hold** the token until it has none. Then it must **re-start** token with value {Y}
      - Rule 2: if *Y* has **no external pointers** in train *sticky-remembered-set* **but *changed-bit*** is ***true,*** it **re-starts** token with value {Y} and changes ***changed-bit* to *false.***
      - Rule 3: if *Y* has **no external pointers** in train *sticky-remembered-set and* ***changed-bit*** is ***false,*** it **relays** the token **unchanged.**
  - token will make, *unchanged,* at most *two* rounds of the ring
    - first one to set *changed-bit* to *false* at every node
    - second one to accomplish garbage detection
    - in Rule 3, it token value is {Y} already, the whole train is found to be garbage and can be reclaimed
      - The token took a complete circle of the ring, with empty sticky-remembered-sets and all changed-bits set to false
    - this acts like a distributed termination algorithm

DGCACY: Train Algorithm for Distributed Systems (4)

- Mutator concurrency (Train *Epochs*)
  - this previous mechanism assumes trains are closed,
    - no new objects may be created in the train
  - hard to guarantee due to *unwanted-relatives*
    - objects in *older* trains, referenced from the *current* train, are moved to *current* train
    - this may create external pointers form *newer* trains that also reference the *older* train.
  - each car in train is associated with either *old* or *new epoch*
    - existing cars when node joins train ring belong to the *old* epoch
    - new cars created in the ring belong to
      - *old* epoch if *changed-bit* was *true*
      - *new* epoch if *changed-bit* was *false*
    - when *changed-bit* **switches** from *false* to *true*
      - **all cars** become member of *old* epoch
  - token-ring performs detection restricted to the *old* epoch
  - when *old* epoch is reclaimed
    - *new* epoch **flips** and becomes the *old* epoch
    - *new* epoch is emptied
    - accomplished by relaying special *end-of-epoch* message

DGCACY: Train Algorithm for Distributed Systems (5)

- Advantages
  - promises to be incremental, asynchronous, mostly concurrent with mutator (only pointer-tracking needs to synchronize with mutator) and complete
- Disadvantages
  - adaptation to distributed scenario introduces high complexity
    - pointer tracking, object substitution
      - high message complexity
    - train management requires that nodes maintain state about garbage detection in course
      - additional complexity to account for cars joining/leaving the train while detection algorithm is running
  - moving objects/cars between trains causes inter-processing messaging
    - the same car, until being reclaimed, may be moved among several trains
    - though, messages are always piggy-backed on regular communication
    - thus, increased GC messaging causes delay in all distributed garbage detection
  - cyclic garbage may delay prompt detection of acyclic garbage in the same train
    - uses the same algorithm for acyclic and cyclic DGC
    - trains may become very large
      - not very important in centralized scenario
      - important in large scale distributed scenario with long chains/rings of processes managing trains
    - one train may contain distributed garbage cycles and many other acyclic garbage elements
      - detection of any may imply/depend on detection of all others
    - eventual co-location of all cyclic garbage is guaranteed eventually
      - however this may take several moves/rounds between trains

DGCACY - DGC with Group Merger (1)

- Detects cycles exclusively comprised within groups of processes [Rodrigues98]
  - improves on DGC in groups [Lang92]
  - multiple groups can merged and synchronized
    - concurrent detections can be re-used and combined
  - fewer synchronization requirements
    - does not try to trace all objects, just those belonging to cycle
- Candidate selection initiates group creation
- Two strictly-ordered distributed phases trace objects
  - mark-red phase
    - marks distributed transitive closure of suspect objects
    - termination of this phase creates a group of processes
  - scan-phase
    - performed independently at each process
    - ensures un-reachability of suspected objects
    - detects objects reachable form other clients outside the group
      - these are marked green, i.e., not garbage
      - green marks must be propagated to other processes
        - » alternate local and remote steps

DGCACY - DGC with Group Merger (2)

- Cycle detector must inspect objects individually
  - strong integration/dependency with execution environment and local GC
- Mutator requests on objects involved in a group detection during mark-green in scan phase can raise race conditions similar to tri-color local tracing
  - to ensure safety, all objects descendents must be atomically marked green
  - mutator is blocked in these situations, when it is actually modifying the objects
- Processes need to store information about group detections in-course comprising them
- When two group detections «meet» they can:
  - merge
  - overlap
  - retreat
    - if one collection in mark-red phase meets a collection already in scan-phase (mark-green)

DGCACY - Mark Propagation with Optimistic Back-tracing (1)

- Processes propagate marks associated with stubs and scions [LeFessant01]
- Marks are complex w.r.t. simple timestamps as in [Hughes85]
  - distance
  - range
  - Generator (scion or local root originating the mark)
  - additional color field
- Local roots and scions are sorted twice according to these marks
  - first, in decreasing order
  - then, in increasing order
- Stubs receive two marks
  - min-max marking, propagates maximum and minimum values found during tracing
  - mark propagation from scions to stubs
  - marks propagate from stub to corresponding scions by messages

DGCACY - Mark Propagation with Optimistic Back-tracing (2)

- Cycle detection is initiated by generators (scions and local roots)
  - generator records are propagated (with fields)
    - creation time
    - range
    - locator of the mark generator
    - color
      - white – pure marks (one generator only)
      - gray – mixing of marks from different generators
      - black – solves optimistic back-tracing errors
- A cycle is detected when:
  - a generator receives back its own mark
  - the mark is white, i.e. un-mixed with marks from other generators
    - if mark is gray, other paths lead to the scion, sub-generations must be initiated

DGCACY - Mark Propagation with Optimistic Back-tracing (3)

- Sub-generations
  - created in the back-trace of the generator that receives gray mark
  - lazy, sequential sub-generations can be very slow
  - optimistic-back-tracing starts back-traces in several processes
    - uses knowledge about sub-generators
    - back-traces are performed in parallel
    - more efficient, yet, unsafe without further cautions
      - special black color in marks advises of generators with revised status
      - ensures safety
- Global approach
  - detects “all” cycles simultaneously
    - avoids need to initiate one detection each specific cycle candidate
    - mark propagation is a global task, continuously performed, with permanent cost
    - should be deferred in tine or executed less frequently
  - imposes specific, heavier LGC
    - must collaborate with the cycles detector
  - tight connection between LGC, acyclic and cyclic DGC
    - inflexibility since prevents different optimizations for each one

DGCACY - DGC-Consistent Cuts in Rotor (1)

- Extends notion of GC-consistent cuts to DGC [Veiga03]
  - developed in the context of Rotor SSCLI [Rotor] (.Net open source version)
  - previous approach found in [Skubi97] addresses centralized systems
- Centralized approach to cycle detection.
  - centralized detection of distributed cycles of garbage spanning process groups
  - eliminates limitations found in [Liskov86]
    - does not require global clock synchronization.
    - does not impose bounded message latency
- Combines snapshots of memory graphs from processes.
    - snapshots can be taken at uncoordinated times.
    - conservative approach ensures safety.
  - graph summarization
    - determines graph border.
    - “inner objects” are discarded.
- Several detectors can be combined hierarchically.

DGCACY - DGC-Consistent Cuts in Rotor (2)

- Distributed GC-Consistent Cut
  - inconsistent cut of a distributed system.
    - combination of several uncoordinated snapshots.
    - useless for common purposes
      - still useful for distributed garbage cycle detection
  - conservative approach ensures safety
    - processes keep list of “youngest” scions created in other processes
    - allows restoration of causality
      - scions are always older than corresponding stubs
    - “younger” scions without stub counterpart become GC-roots
    - references from “unknown” processes become GC-roots
  - only detects cycles fully enclosed in a GC-cut
    - liveness depends on receiving snapshots from processes

DGCACY - DGC-Consistent Cuts in Rotor (3)

- Graph Summarization (Graph Description Reduction)
  - regards object graphs as opaque spaces.
    - only remote references are considered.
      - incoming – scions.
      - outgoing – stubs.
      - also accounts for local reachability of stubs
  - determines associations among scions and stubs
    - for each scion, computes a set of stubs reachable from it.
      - optimized so that each object is not analyzed several times.
      - related to the technique used in [Mahe97]
    - performed off-line.
      - infrequent, incremental, uncoordinated, no disruption to applications.
  - complete for DGC use, provides great reduction in size
  - minimizes detector complexity and demands w.r.t.:
    - CPU load
    - bandwidth usage

DGCACY - DGC-Consistent Cuts in Rotor (4)

- Cycle Detection Process (CDP)
  - logically centralizes cycle detection
  - receives the summarized graph snapshots from participating processes
  - restores causality among scions and corresponding stubs
    - if necessary, some scions are promoted to CDP roots
  - performs Mark-Sweep on summarized graphs stating from CDP root-set that includes:
    - stubs reachable-locally in their processes
    - scions promoted to CDP roots
  - identifies scions, with their corresponding stub also present in the snapshots, that were not marked
    - these scions are live for the acyclic DGC
    - yet, they are not reachable from the CDP roots
    - thus, they belong to distributed cyclic garbage
  - instruct processes to delete scions holding the cyclic garbage
    - special delete.scion message
    - breaks cyclic garbage into acyclic garbage.
    - no intrusion with mutator, since these objects are already unreachable to it
  - acyclic DGC will then reclaim remaining garbage stubs and scions
  - local GC will actual reclaim objects after
  - results can be sent to upper-level CDP

DGCACY - DGC-Consistent Cuts in Rotor (5)

- Summary
  - notion of CG-Consistent Cut extended to DGC.
  - comprehensive solution to DGC.
    - reference-listing DGC running on Rotor [Rotor].
    - centralized cycles detector.
      - no global synchronization.
      - scalable.
      - makes progress without requiring participation of all processes.
      - does not interfere with local collectors
        - » does not require re-tracing of objects belonging to cycle being detected
        - » does not need sequential distributed phases
      - processes need not store state about detections in course
        - » fault-tolerance
  - shares drawbacks with other centralized approaches
    - CDP availability and performance may become bottleneck
      - even though several CDP may exist for intersecting sets of processes

DGCACY - Algebra-based Distributed Cycle Detection (1)

- De-centralized version of DGC-Consistent-Cuts [Veiga05]
  - avoids need of dedicated processes/nodes to detect cycles
  - graph summarization information is bi-directional
    - ScionsToStubs – set of stubs reachable from a scion
    - StubsToScions – set of scions that lead to a stub
      - this is directly obtained as the previous is being calculated
- Processes forward cycle detection messages (CDM)
  - piggy-backed on acyclic DGC messages
  - conceptually targeting an object belonging to the cycle being detected
  - carry algebraic representation of cycle candidate graph being detected
    - algebra comprises two sets
      - Source-set
      - Target-set
- Upon CDM reception processes:
  - apply matching predicate to CDM algebra and either
    - forward updated version of CDM
    - terminate detection

DGCACY - Algebra-based Distributed Cycle Detection (2)

- Algebraic reduction of CDM (Cycle-Found?)
  - matches elements of source and target-sets.
  - elements present in both sets are cleared.
  - if source-set becomes empty → Cycle Found.
    - instruct scion deletion (triggers reclamation of the rest of the cycle).
  - remaining elements in source-set are unresolved dependencies.
- CDM to be forwarded
  - stubs *reachable from* scion receiving message.
    - inserted in target-set.
  - additional scions *targeting* stubs in target-set.
    - inserted in source-set as extra-dependencies to be resolved.
  - new CDMs lazily sent to processes holding targeted objects.
    - piggybacked on DGC messages or queued/sent in batch.

DGCACY - Algebra-based Distributed Cycle Detection (3)

- Cycle Detection Algebra
  - representation of in-course cycle detection
    - carried in **C**ycle **D**etection **M**essages
    - detection state in CDM, not stored in participating processes
    - initiated on a cycle candidate (heuristically determined)
    - lazily forwarded among processes comprising path being tested

      { Source-set } → { Target-set }
  - Source-set
    - accounts for dependencies, i.e., scions converging to the path being detected and still not found to belong to garbage
  - Target-set
    - accounts for target objects, stubs, the CDM has been forwarded to
  - CDM reception at a process triggers CDM Matching
    - may detect cycle, forward updated CDM, or abandon detection

DGCACY - Algebra-based Distributed Cycle Detection (4)

- Incremental DGC Consistent-cut (CDM Graphs)
  - inconsistent cut of a distributed system
    - combination of several uncoordinated snapshots
    - useless for common purposes
  - conservative approach ensures safety w.r.t. cycle detection
    - pair-wise combination of processes through snapshots and CDMessages
    - stub without matching scion → not current enough, ignore CDM
    - scion without matching stub → CDM is never sent
    - only detects cycles fully enclosed in a CDM-Graph
    - liveness depends on snapshots update from processes
      - cycles always get older, eventual detection
  - concurrency with mutator
    - scions and matching stubs piggyback counters on remote invocations
      - different values denounce mutator activity on the CDM-Graph after one of the snapshots was taken

DGCACY - Algebra-based Distributed Cycle Detection (5)

- Summary
  - properties inherited from dgc-consistent cuts
    - no global synchronization required
    - scalable
    - makes progress without requiring participation of all processes
  - allows different optimization techniques for
    - LGC
    - DGC
    - Distributed cycle detection
  - improves on previous work
    - Cycle Detection Algebra
    - incremental construction of DCG-Consistent Cuts for cycle detection (CDM Graphs)
    - processes need not store state about cycle detections in course
      - safety preserved if cycle detections are
        - » stopped, delayed, repeated,
      - several detections may be safely performed simultaneously
        - » over the same processes or even for the same cycle

Garbage Collection in Transactional Systems (GCTS)

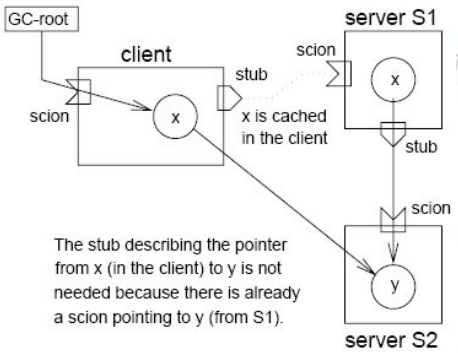
- transactional reference listing
- transactional mark and sweep
- atomic copy
- replicated copy
- GC-cuts

GC in Transactional Systems (GCTS)

- These algorithms are extensions of the reference counting and tracing algorithms presented in the previous sections
  - such extensions are needed to deal with the specific safety problems posed by transactional systems
- We describe the following:
  - transactional reference listing
  - transactional mark and sweep
  - atomic copy
  - replicated copy

GCTS - Transactional Reference Listing (1)

- We assume an object-oriented client server database with multiple storage servers Thor [Liskov92]:
  - clients cache in main memory objects that are being accessed
- When a server recovers from a failure:
  - it must retrieve all the scions pointing to objects allocated in the server partitions
  - scions whose stubs are held by other servers are easily recovered because such scions are kept in stable storage
  - scions whose stubs are held by clients are not kept in stable storage because that would be too expensive
    - instead, a server keeps in stable storage a list of its clients.
  - thus when a server recovers:
    - it knows who its clients are and sends them query messages asking for their stubs.



GCTS - Transactional Reference Listing (2)

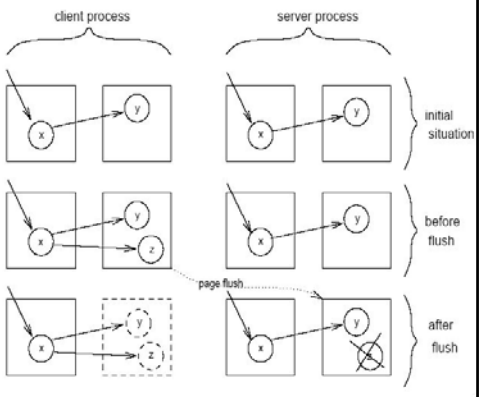
- If a client process has not communicated with a server for a long time and does not respond to repeated query messages:
  - the server assumes it has failed
- However the client may just be unable to communicate with that server because of network problems
  - it might happen that the client communicates with other servers
  - imagine that a server S1 assumes that a client has failed while a second server S2 does not (figure in previous slide)
  - then, the first server discards the scions whose stubs were held by the client
  - this may cause the second server to unsafely reclaim an object y that is still reachable from the client
  - this erroneous behavior is due to the inconsistent views the servers have about the client
- To solve this problem:
  - it is necessary that all servers get a consistent view of a client status
  - for this purpose there is an atomic shutdown protocol
  - once this protocol is executed no server will honor requests from the client that has been shutdown
  - thus, in the previous example S1 and S2 would agree that the client has crashed and y would be safely reclaimed because the client would not be allowed to follow the pointer from x to y
- A problem with this solution is that the shutdown protocol is not easily scalable

GCTS - Transactional Mark-and-Sweep (1)

- This algorithm is a variant of the basic uniprocessor mark-and-sweep:
  - with the partitioned and incremental functioning modes extended in order to cope with transactions in a client-server database
  - the collector runs on the server
  - it was implemented in the EXODUS database [Carey86]
- When a reference to some object is discarded inside a transaction:
  - the pointed object is eligible for collection only after the commit of that transaction
- This rule prevents:
  - the unsafe reclamation of an object that becomes unreachable during a transaction that will later abort
  - in fact, the abort of the transaction will make that object reachable again
  - thus, any object that might become unreachable during a transaction will only be reclaimed if it remains unreachable after the commit of the transaction

GCTS - Transactional Mark-and-Sweep (2)

- An object created during a transaction is eligible for reclamation only:
  - after the commit of the transaction that created it
  - this rule prevents
    - the unsafe reclamation of reachable objects due to the flush of pages from the client to the server in an order that is not controlled by the collector
- E.g.:
  - x is reachable from the GC-root
  - the application creates object z and makes it reachable from x
  - then, the page containing objects y and z is flushed to the server
  - when the collector runs in the server only the page containing objects y and z has been flushed
  - thus, z is unsafely reclaimed because it is not reachable from x



GCTS - Transactional Mark-and-Sweep (3)

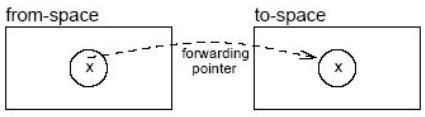
- The memory space reclaimed by the sweeping phase can be reused for allocating new objects:
  - only when the freeing of that space is reflected in stable storage
- This ensures that:
  - during a recovery after a failure there will always be enough space for allocating objects
- This is illustrated by the following example:
  - suppose that a client creates objects x, y and z in a page that had previously been garbage collected at the server
  - however that page has not been written to stable storage yet
    - on disk it still contains a garbage object t (not yet swept)
  - now the client commits the transaction
    - as a result the page and the associated log are sent to the server
    - the log is written to stable storage and the system crashes before the updated page reaches the disk
  - then recovery starts
    - the problem is that objects x, y and z must be created (according to the log), but
    - there is not enough space as the page that is used for the recovery (the one on disk) still contains the garbage object t (not yet swept)

GCTS - Atomic Copy (1)

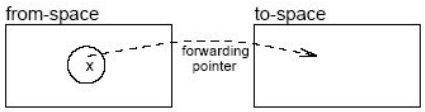
- This algorithm [Detlefs91,Kolodner93] is based on Bakers copy GC:
  - it works on a single partition
  - both functioning modes GC-only and incremental can be found in the literature
- Reachable objects are moved from from-space to to-space and patched accordingly
  - this may interfere with the stability of the store as explained now
- Imagine there is a failure while the collector is running
  - then, during the failure recovery
    - the GC algorithm must find which objects have already been moved, and which objects have already been patched
    - otherwise the pointer graph will be corrupted

GCTS - Atomic Copy (2)

- Suppose there is a failure after object x has been moved to to-space:
  - a forwarding pointer was left in from-space, and
  - only the from-space has been written to disk
- Thus, after the failure the disk will not contain a valid version of x:
  - the version in from-space has been overwritten by the forwarding pointer, and
  - the version in to-space is not available on disk
- If the collector were to be restarted after the failure:
  - x would never be moved again to to-space, and
  - the pointer graph would be corrupted



(a) virtual memory before failure



GCTS - Atomic Copy (3)

- The solution to this problem consists of:
  - making the copy algorithm atomic
- This requires that the recovery of a failure puts from-space and to-space in a state from which the collection can continue
  - this is done by applying the write ahead log protocol [Bernstein87] to the disk contents
  - for this purpose both the move and patch of each reachable object must be logged
  - this solution even after being optimized remains costly
  - in the next slide we describe the un-optimized version (incremental GC)

GCTS - Atomic Copy (4)

- The move of a reachable object proceeds as follows:
  - pin the from-space object that will be overwritten by the forwarding
  - pointer
  - move the object to to-space and insert the forwarding pointer in its from-space version
  - create a log entry with the from-space and the to-space addresses of the object
    - create another log entry with the to-space address of the object and its contents
  - now the move of the object is completed
  - after both log entries are written to disk unpin the from-space object
- The scan and patch of a to-space object proceeds as follows:
  - pin the object to be scanned
  - scan the object and patch its from-space pointers to to-space pointers moving the pointed objects as needed
  - create a log entry with the object contents
  - unpin the object after the log entry is written to disk

GCTS - Atomic Copy (5)

- Concerning correctness:
  - the GC algorithm must not consider to be unreachable an object x that became so due to a transaction that is still running
- If the mentioned transaction aborts:
  - object x is now reachable again as it was before the transaction has started
- Thus:
  - the collector has to consider as part of the GC-root the log of running transactions
- Note that the collector could run as a single long user level transaction or a series of short transactions:
  - however in the first case this would lead to a long GC pause time which is highly disruptive
  - in the second case it could easily lead to deadlock or to a situation in which a long user transaction could prevent the collector from making progress
  - for these reasons the collector runs at a level below the transactions support

GCTS - Replicated Copy

- The replication based copy collector [O'Toole93] is particularly well suited to single-site transactional systems:
  - this is due to the fact that while the collector moves objects to to-space the mutator continues to access their from-space replicas rather than the replicas in to-space
- When every reachable object has been moved to to-space:
  - a flip is performed, and
  - the mutator then starts accessing the to-space replicas
- If a process fails while the collector is running before the flip:
  - the collector simply restarts with the recovered from-space
  - all GC operations already done are lost

GCTS – GC-Cuts In Object-Oriented Databases (1)

- GC developed for the O_2 Object-Oriented Database System [Skubi97]
- Based on the premise that only reference «overwriting» can create garbage
  - detects these operations with resort to transactional synchronization mechanisms
    - read/share and write/exclusive locks on pages
- Garbage collector constructs GC-cuts
  - a set with at least one copy of every page in the database
    - may hold several copies of the same page if necessary
  - copies of pages are created at different instants
    - this may avoid mutator disruption
  - copies of pages may be inconsistent
    - they may hold different content valid during different transactions
  - combination with knowledge from locks ensures consistency
    - if a page in the cut contains references to objects in another page, the latter must also be included in the cut

GCTS – GC-Cuts In Object-Oriented Databases (2)

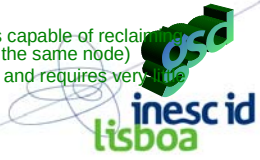
- CG-cuts are constructed incrementally in parallel with the mutator
- When CG-cut is complete, garbage can be identified and reclaimed
  - mark and sweep phase are sequential but concurrent with mutator
- An object is considered garbage if and only if it is garbage in every page in the CG-cut where it occurs
- Applied exclusively in the context of centralized object databases
  - potential size of complete cut makes it inadequate to distributed environments

Distributed Garbage Collection in Replicated Memory Systems (DGCARM)

- GC and DSM consistency
- DGC on replicated memory
- DGC for wide area replicated memory
- Complete and scalable DGC

DGCARM – Garbage Collection and DSM Consistency in the BMX System (1)

- The BMX system [Ferreira94] is a software platform providing:
  - persistent weakly consistent shared distributed virtual memory, and
  - copying garbage collection
- The main goal of the design of the BMX DGCA is:
  - to minimize the communication overhead due to collection between nodes of the system, and
  - to avoid any interference with the DSM memory consistency protocol
- The BMX DGCA design is based on the fundamental observation that:
  - in a weakly consistent DSM system the memory consistency requirements of the garbage collector are less strict than those of the applications
- Thus, the garbage collector reclaims objects independently of other replicas of the same objects without interfering with the DSM consistency protocol:
  - this is a relevant issue as the interference between the garbage collector and the consistency protocol could potentially nullify the advantages of using a weakly consistent DSM system
- E.g. when updating a reference inside an object to reflect the new location of a live descendent that has already been copied:
  - the garbage collector should not require exclusive write-access to modify the object
  - if exclusive write-access was needed, read-access to all other replicas of the object would have to be invalidate, therefore nullifying the advantage of using weak consistent DSM
- Furthermore:
  - the collector does not require reliable communication support and is capable of reclaiming distributed cycles of dead objects (as long as they are all cached in the same node)
  - the collector does not interfere with applications' consistency needs and requires very little synchronization/communication



DGCARM – Garbage Collection and DSM Consistency in the BMX System (2)

- In the BMX system:
  - objects are allocated in bunches (set of non-overlapping segments) in a single 64-bit address space spanning the whole network, including secondary storage
  - objects are kept weakly consistent by the entry consistency protocol
- The DGCA has the main characteristics:
  - a cached copy of a bunch can be collected independently of any other copy of that same bunch on other nodes
  - only locally-owned live objects are copied by a bunch garbage collector; not owned live objects are simply scanned, and
  - references to copied objects are lazily updated, either by taking advantage of messages sent on behalf of the consistency protocol (piggy-backing), or in the background
  - in any circumstance, the garbage collector acquires neither a read nor a write token

DGCARM – Garbage Collection and DSM Consistency in the BMX System (3)

- Each bunch of objects is garbage collected independently of any other bunch:
  - thus, each cached copy of a bunch holds a stub table and a scion table containing information about outgoing and incoming references, respectively
  - the bunch garbage collector (BGC) executes the collection on a local replica of a bunch independently from the collection of any other bunch and other replicas of the same bunch
- The BGC is based on the algorithm by O'Toole [O'Toole93] for three main reasons (however, any other algorithm could be used):
  - the flip time is very small and therefore not disruptive to applications
  - portability (no virtual memory manipulations), and
  - objects are non-destructively copied (suitable for recovery purposes)
- The BGC does not interfere with applications' consistency needs, neither when a live object is copied from from-space to to-space, nor when updating references to the object's new location:
  - a live object is copied to to-space only at its owner node and its new address is written in the object's header (forwarding pointer mechanism)
  - this modification is strictly local to the owner node, therefore it does not imply acquiring the corresponding DSM write token
  - otherwise (object not locally owned), the object is simply scanned; the scanning does not have to be done in a consistent replica of the object
  - it is not necessary to acquire the corresponding DSM read token

DGCARM – Garbage Collection and DSM Consistency in the BMX System (4)

- After being moved to to-space, all the references pointing to the object that has just moved must be updated:
  - these references may be spread across many bunches
  - however, this does not interfere with the consistency protocol (no need to acquire the write token of the objects containing such references)
- The fundamental point that allows the mentioned updating (on the source objects) to be done without interfering with the consistency protocol is twofold:
  - for objects with cached replicas on the owner node (where the target object has been copied), such an update can be made without acquiring the write token, and
  - for objects replicated on other nodes, the update can be made by piggy-backing the new location of the moved object on inter-node DSM acquire/release messages sent on behalf of applications

DGCARM – Distributed Garbage Collection on Replicated Memory – Larchant (1)

- Larchant [Ferreira96] is a cached distributed shared store:
  - based on the model of a DSM with persistence by reachability
  - data is replicated in multiple sites for performance and availability
  - reachability is assessed by tracing the pointer graph by means of a hybrid DGC algorithm
- The DGC in Larchant is a hybrid or partitioned algorithm as it combines tracing within a partition with reference counting across partition boundaries:
  - each process may trace its own replicas independently of one another and of other replicas
  - counting at some process is asynchronous to other processes and asynchronous to the local mutator
  - in addition, counting is deferred and batched
- Both the tracing and the distributed counting garbage collector run independently of coherence:
  - garbage collection does not need coherent data
  - never causes coherence messages nor input/output
  - it does not compete with applications locks or working sets
  - coherence messages must at times be scanned before sending

DGCARM – Distributed Garbage Collection on Replicated Memory – Larchant (2)

- The DGC in Larchant works according to five safety rules:
  - they guarantee the DGC correctness [Ferreira98]
- These safety rules are minimal and generally applicable given the asynchrony to applications and the minimum assumptions made concerning coherence:
  - Union, Increment before Decrement, Comprehensive Tracing, Clean Propagation, Causal Delivery
- Union Rule:
  - an object may be reclaimed only if it is unreachable from the union of all replicas (of the pointing objects)
- Increment before Decrement Rule:
  - when an object is scanned, the corresponding "increment" messages (reference-counting) must be sent immediately (i.e., put them in the sending queue)
- Comprehensive Tracing Rule:
  - when a "union" or a "decrement" message (reference-counting) is sent, all replicas (on the sending site) must be GC-clean
- Clean Propagation Rule:
  - an object must be scanned before being replicated
- Causal Delivery Rule:
  - DGC messages must be delivered in causal order

DGCARM – DGC on Wide Area Replicated Memory (1)

- Acyclic, scalable DGC for wide area replicated memory systems (WARM) [Sánchez01]
- Safe w.r.t. handling replicated data correctly
  - processes only manipulate replicated objects locally
    - no remote invocations
  - multiple replicas place at different processes may be edited simultaneously
  - each process manages InProp and OutProp Lists
    - InProp entries indicate the process from which each object has been replicated
    - OutProp entries indicate processes to which each object has been replicated
    - account for incoming and outgoing object replicas
    - when replica is no longer reachable locally
      - Unreachable message is sent to originating process and USent bit set in InProp
      - URcvd bit of sender process is set in corresponding OutProp
    - when all replicas are unreachable, master replica instructs deletion
      - when all URcvd bit are set, a delete message is sent lazily to every process
      - upon arrival of delete message, InProp is deleted
      - replicated object is then at mercy of the Local GC

DGCARM – DGC on Wide Area Replicated Memory (2)

- Improvement over Larchant
  - different implementation of the union rule
  - avoids need for causal message delivery
- Scalable to large-scale distributed systems and orthogonal to coherence mechanisms
  - non-intrusive with coherence engine
  - needs only be advised of replica propagation operations
  - is safe regardless of coherence mechanisms used
- Unable to detect distributed cycles comprising replicas
  - cycles in WARM are potentially more frequent than without replication
    - only a single replica of an object needs to be involved in distributed cycle to prevent reclamation of every other replica
  - cycles in WARM waste more storage as replication factor grows
    - algorithm completeness being addressed in current work [Veiga05a]

DGCARM - Comprehensive Approach for Memory Management of Replicated Objects

- Addressing DGC completeness on replicated objects [Veiga05a]:
  - detects distributed garbage cycles comprising replicated objects spanning several processes
  - extension to DGC-WARM [Sánchez01] with DGC-Consistent-Cuts [Veiga03] made replication-aware
    - first algorithm for replicated objects that is complete
      - safety w.r.t. replicated objects from DGC-WARM
      - other properties shared with DGC-Consistemt-Cuts
  - enforces union rule in dgc-consistent-cuts
  - detects distributed cycles comprised within groups of processes
  - several DCDP cooperate (hierchically, or otherwise)
    - no negotiation/synchronization needed among pariticipating processes

Concluding Remarks

- performance issues
- tracing and reference-counting unification
- final remarks

Performance Issues in Uniprocessor Garbage Collection Algorithms

- Myths and realities [Blackburn04]
  - GC is better than no-GC
    - GC offers programming soundness
    - performance benefits over Manual Memory Management
      - GC contiguous allocation out-performs free-list allocation
        - » architectural trends accentuate this advantage in the future
    - out-performs No-MM at all
      - locality benefits.
        - » GC maintains live objects close to each in other in memory
        - » manual MM and No-MM degrade object locality
      - prevents memory exhaustion
  - generational collectors perform widely better
    - reduced collection time (opposed to whole-heap GC)
      - mark-and-sweep degrades with large-heaps
    - further improved locality

Tracing & Reference Counting Unification(1)

- Tracing and Reference-counting are duals [Bacon04]
  - tracing operates on live objects (*matter)*
    - transversal starts with mutator roots and detects all live objects
    - tracing can be regarded as a one-bit "sticky" reference count
      - initializes objects reference-count (RC) to zero and increments them when they are referenced from live objects
    - extra sweep phase to collect dead objects
      - may be optimized with semi-space copying collectors (time-space trade-off)
  - reference-counting operates on dead objects (*anti-matter)*
    - transversal starts with anti-roots (objects whose reference-count has been decremented) and detects further downstream objects whose RC will reach zero
      - this is the graph complement of live objects except cycles
      - in each iteration, RC of objects are >= to the *real value*. therefore, GC decrements them when they are referenced from garbage objects
    - extra-phase to detect cycles (*trial deletion using RC*)
- High performance GC algorithms are all hybrids of some kind
  - even when it this *hybridization* is not trivial
  - deferred reference counting
    - Zero Count Tables are tracing elements
  - generational GC
    - Remembered Sets are RC elements
      - set-representation of non-zero reference-counts
      - complementary to a ZCT

Tracing & Reference Counting Unification (2)

	Tracing	Reference Counting
Starting Point	Mutator Roots	Anti-Roots zero-RC objects
Transversal	Forward	Forward
Transversed Objects	Live	Dead
Initial RefCount	Low (0)	High
RefCount Convergence	Addition	Subtraction
Extra Phase	Sweep phase	Trial deletion

Final Remarks (1)

- The issue of garbage collection (GC) is a challenging topic
- GC is both a mature and dynamic research area
  - early works date around 40 years ago
  - nonetheless, still a field of active and relevant research nowadays
- GC is not only research
  - industry adoption has been steadily increasing
    - LGC in most script languages
    - DGC in Java and .Net Remoting
- GC is a cross-cutting issue in systems and research
  - LGC deals with programming correctness and system performance
  - DGC research motivates and exercises key aspects of distributed algorithm design
    - safety
    - completeness
    - liveness
    - asynchrony
    - scalability
    - termination
  - 3-tiered approach to GC (local, acyclic and cyclic DGC) further leverages the previous aspects
- Thus, GC is interesting in a theoretical as well as practical perspective

Final Remarks (2)

- Many problems have been solved along the years but some still remain, e.g.:
  - the completeness of distributed garbage collection algorithms is still a research issue
  - the issues of performance of garbage collectors and their adaptation to resource-constrained environments also deserves an important research effort
- Garbage collection (GC) automatically ensures referential integrity therefore:
  - improving program reliability, and
  - programming productivity
  - in addition, it may compact memory thus reducing fragmentation and improving locality
- Both in centralized and in distributed systems, the nightmare of manual memory management increases as the number of objects, references, nodes and users scales up:
  - then, GC becomes indispensable !!!

References

References (A)

- [Ali84] K. A. Ali. Object Oriented Storage Management and Garbage Collection in distributed Processing Systems, TRITA-CS-8406, Stockholm, Sweden, Dec. 1984.
- [Appel88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation, pages 11--20, Atlanta, Georgia, June 1988. ACM Press.
- [Appel89] Andrew W. Appel. Runtime tags aren't necessary. Lisp and Symbolic Computation, 2:153--162, 1989.
- [Appel89a] Andrew W. Appel. Simple generational garbage collection and fast allocation. Software Practice and Experience, 19(2):171--183, February 1989.

References (B1)

- [Bacon01] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In Proceedings of SIGPLAN 2001 Conference on Programming Languages Design and Implementation, ACM SIGPLAN Notices, Snowbird, Utah, June 2001. ACM Press.
- [Bacon04] David F. Bacon, Perry Cheng, V. T. Rajan: A unified theory of garbage collection. Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada. ACM 2004
- [Baker78] Henry G. Baker, Jr. List processing in real time on a serial computer. Communications of the ACM, 21(4):280--294, April 1978.
- [Bartlet88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988
- [Bernstein87] Philip Bernstein, Vassos Hadzilacos and Nathan Goodman editors. Concurrency Control and Recovery in Database Systems. Addison Wesley, Massachusetts, 1987.
- [Bevan87] D. I. Bevan. Distributed garbage collection using reference counting. Parallel Arch. and Lang. Europe, pages 117--187, Eindhoven (The Netherlands), June 1987. Springer-Verlag Lecture Notes in Computer Science 259.
- [Birman91] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. ACM Transactions on Computer Systems, 9(3):272--314, August 1991.
- [Birrell93] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. Proceedings of the 14th ACM Symposium on Operating Systems Principles, pages 217--230, Asheville, NC (USA), December 1993.

References (B2)

- [Bishop77] Peter B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. PhD thesis, Massachusetts Institute of Technology Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR-178.
- [Blackburn03] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.
- [Blackburn04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and reality: The performance impact of garbage collection. In Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems, New York, NY, June 2004
- [Bohem91] Hans J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation, pages 157--164.
- [Bohem91a] Hans-Juergen Boehm and David Chase. A proposal for garbage-collector-safe compilation. The Journal of C Language Translation, 4(2):126--141, December 1991.
- [Bohem93] Hans-Juergen Boehm. Space-efficient conservative garbage collection. Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation, pages 197--206, Albuquerque, New Mexico, June 1993. ACM Press.
- [Brooks84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pages 108--113.

References (C)

- [Carey86] Michael Carey and David DeWitt. The architecture of the EXODUS extensible DBMS. Proc. Int. Workshop on Object Oriented Database Systems, 52— 65. Pacific Grove (USA), September 1986.
- [Cardelli89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalso, and Greg Nelson. Modula-3 report (revised). Research Report 52, Digital Equipment Corporation Systems Research Center, November 1989.
- [Cheney70] C. J. Cheney. A non-recursive list compacting algorithm. Communications of the ACM, 13(11):677--678, November 1970.
- [Christopher84] Thomas W. Christopher. Reference count garbage collection. Software Practice and Experience, 14(6):503--507, June 1984.
- [Cohen81] Jacques Cohen. Garbage collection of linked data structures. Computing Surveys, 13(3):341--367, September 1981.

References (D)

- [Demers90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, pages 261--269, San Francisco, California, January 1990. ACM Press.
- [Detlefs91] David Detlefs. Concurrent Atomic Garbage Collection. PhD Thesis Dept. of Computer Science, Carnegie Mellon University. Pittsburgh (Pennsylvania). November 1991. Technical report CMU-CS-90-177.
- [Detlefs92] David Detlefs. Garbage collection and run-time typing as a C++ library. C++ Conference, pages 37--56, Portland (USA), August 1992. Usenix.
- [Deutsch76] L.Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. Communications of the ACM, 19(9):522--526, September 1976.
- [Dickman91] Peter Dickman. Distributed Object Management in a Non-Small Graph of Autonomous Networks with Few Failures. Ph.D. thesis. University of Cambridge, UK, 1991.
- [Dijsktra78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. Communications of the ACM, 21(11):966--975, November 1978.
- [Diwan92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed language. Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation, pages 273--282, San Francisco, California, June 1992. ACM Press.

References (E-F)

- [Edelson92] Daniel R. Edelson. Precompiling C++ for garbage collection. Proc. 1992 International Workshop on Memory Management, pages 299--314, Saint-Malo (France), September 1992. Springer-Verlag.
- [Edelson92a] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. C++ Conference, pages 1--19, Portland, (USA), August 1992. Usenix.
- [Ellis93] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. Technical Report 102, Digital Equipment Corporation Systems Research Center, 1993.
- [Fenichel69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. Communications of the ACM, 12(11):611--612, November 1969.
- [Ferreira91] Paulo Ferreira. Reclaiming storage in an object oriented platform supporting extended C++ and Objective-C applications. Proc. of the International Workshop on Object Orientation in Operating Systems, Palo Alto (USA), October 1991.
- [Ferreira94] Paulo Ferreira and Marc Shapiro. Garbage collection and DSM consistency. In First Symposium on Operating Systems Design and Implementation, pages 229-241, Monterey, CA, November 1994. ACM Press
- [Ferreira96] Paulo Ferreira and Marc Shapiro. Larchant: Persistence by Reachability in Distributed Shared Memory through Garbage Collection. Proceedings of the International Conference on Distributed Computing Systems (ICDCS'96) Hong Kong. May 1996.
- [Ferreira98] Paulo Ferreira and Marc Shapiro. Modelling a distributed cached store for garbage collection: the algorithm and its correctness proof. European Conf. on Object-Oriented Programming (ECOOP'98), Brussels (Belgium), July 1998.
- [Fisher82] Michael J. Fischer, Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In Proceedings of the 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Los Angeles, California, 1982

References (G)

- [Goldberg89] Benjamin Goldberg. Generational reference counting: A reduced-communication distributed storage reclamation scheme. Programming Languages Design and Implementation, number 24(7) in SIGPLAN Notices, pages 313--321, Portland, (USA), June 1989. SIGPLAN, ACM Press.
- [Goldberg91] Benjamin Goldberg. Tag-free garbage collection for strongly-typed programming languages. Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation [98], pages 165--176.
- [Gupta93] Aloke Gupta and W. K. Fuchs. Garbage collection in a distributed object-oriented system. IEEE Transactions on Knowledge and Data Engineering, 5(2), April 1993.

References (H)

- [Hayes91] Barry Hayes. Using key object opportunism to collect old objects. Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA'91), pages 33--46, Phoenix, Arizona, October 1991. ACM Press.
- [Hudak82] Paul Hudak, Simon L. Peyton Jones, and Phillip Wadler. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). ACM SIGPLAN Notices, 27(5), May 1992.
- [Hudson92] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects Yves Bekkers and Jacques Cohen, editors. Proceedings of International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, St. Malo, France, 16-18 September 1992. Springer-Verlag.[Hudson97] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications – 12th Annual Conference, volume 32(10) of ACM SIGPLAN Notices, Atlanta, GA, October 1997. ACM Press.
- [Huelsbergen93] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), pages 73--82, San Diego, California, May 1993. ACM Press. Published as SIGPLAN Notices 28(7), July 1993.
- [Hughes85] R. John M. Hughes. A distributed garbage collection algorithm. In Jouannaud [FPCA85], pages 256-272

References (J-K-L)

- [Juul92] Neils-Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In International Workshop on Memory Management, volume 637 of Lecture Notes in Computer Science, St. Malo, France, 16-18 September 1992.
- [Kolodner93] E. Kolodner and W. Weihl. Atomic incremental garbage collection and recovery for large stable heap. Proc. of the ACM SIGMO, 177—185, Washington DC (USA), June 1993.
- [Ladin92] Rivka Ladin and Barbara Liskov. Garbage collection of a distributed heap. In International Conference on Distributed Computing Systems, Yokohama, June 1992.
- [Lamport78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558--565, July 1978.
- [Lang92] Bernard Lang, Christian Quenniac, and José Piquer. Garbage collecting the world. In POPL [POPL92], pages 39-50. Available here.
- [LeFessant01] Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In Principles of Distributed Computing (PODC), Rhodes Island, August 2001.
- [Lieberman83] 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.
- [Liskov86] Barbara Liskov and Rivka Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In J. Halpern, editor, Proceedings of the Fifth Annual ACM Symposium on the Principles on Distributed Computing, pages 29-39, Calgary, August 1986. ACM Press.
- [Liskov92] Barbara Liskov, Mark Day and Liuba Shrira Distributed object management in Thor. Proc. Int. Workshop on Distributed Object Management, 1—15. Edmonton (Canada), August 1992.
- [Louboutin97] Sylvain R.Y. Louboutin and Vinny Cahill. Comprehensive distributed garbage collection by tracking causal dependencies of relevant mutator events. In Proceedings of ICDCS'97 International Conference on Distributed Computing Systems. IEEE Press, 1997

References (M-1)

- [Mahe94] Umesh Maheshwari and B. Liskov. Fault-tolerant distributed garbage collection in a client-server, object database. Proceedings of the Parallel and Distributed Information Systems, pages 239--248, Austin, Texas (USA), September 1994.
- [Mahe95] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by controlled migration. In Proceedings of PODC'95 Principles of Distributed Computing, 1995. Later appeared in Distributed Computing, Springer Verlag, 1996
- [Mahe97] Umesh Maheshwari and Barbara Liskov. Collecting cyclic distributed garbage by back tracing. In Proceedings of PODC'97 Principles of Distributed Computing, pages 239-248, Santa Barbara, CA, 1997. ACM Press
- [Mahe97a] Umesh Maheshwari and Barbara Liskov. Partitioned garbage collection of a large object store. In Proceedings of SIGMOD'97, 1997.
- [McCarthy60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. Communications of the ACM, 3(4):184--195, April 1960.

References (M-2)

- [Minsky63] Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I. Memo 58, Massachusetts Institute of Technology Project MAC, Cambridge, Massachusetts, 1963.
- [Moon84] David Moon. Garbage collection in a large Lisp system. Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, pages 235--246.
- [Moss89] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. Proceedings of the Second International Workshop on Database Programming Languages, Gleneden Beach, (USA), June 1989. Also available as COINS Technical Report 89--68, Object-Oriented Systems Laboratory, Dept. of Computer and Information Science, University of Massachusetts, Amherst.
- [Moss96] J. Eliot B. Moss, David S. Munro, and Richard L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In Proceedings of the Seventh International Workshop on Persistent Object Systems. Morgan Kaufmann, June 1996.

References (N-O-P-Q)

- [Nettles92] S. Nettles, J. O'Toole, D. Pierce, and N. Haines. Replication-based incremental copying collection. Proc. Int. Workshop on Memory Management, number 637 in Lecture Notes in Computer Science, pages 357--364, Saint-Malo (France), September 1992. Springer-Verlag.
- [O'Toole93] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. Proceedings of the Fourteenth Symposium on Operating Systems Principles, Asheville, North Carolina (USA), December 1993. ACM Press. Published as Operating Systems Review 27(5).
- [Piquer91] José M. Piquer. Indirect reference-counting, a distributed garbage collection algorithm. PARLE'91---Parallel Architectures and Languages Europe, volume 505 of Lecture Notes in Computer Science, pages 150--165, Eindhoven (The Netherlands), June 1991. Springer-Verlag.
- [Queinnec89] Christian Queinnec, Barbara Beaudoing, and Jean-Pierre Queille. Mark DURING Sweep rather than Mark THEN Sweep. Eddy Odijik, M.Rem, and Jean-Claude Sayr, editors, Parallel Architectures and Languages Europe, number 365, 366 in Lecture Notes in Computer Science, Eindhoven, The Netherlands, June 1989. Springer-Verlag.

References (R)

- [Rana83] S. P. Rana. A distributed solution to the distributed termination problem. Information Processing Letters, 17:43-46, July 1983.
- [Rivera97] Gustavo Rodriguez-Rivera and Vince Russo. Cyclic distributed garbage collection without global synchronization in CORBA. In Peter Dickman and Paul R. Wilson, editors, OOPSLA '97 Workshop on Garbage Collection and Memory Management, October 1997.
- [Rodrigues98] Helena C. C. D. Rodrigues and Richard E. Jones. Cyclic distributed garbage collection with group merger. In Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98, volume 1445 of Lecture Notes in Computer Science, Brussels, July 1998. Springer-Verlag., pages 249-273. Also UKC Technical report 17-97, December 1997.
- [Rotor] J Whittington . Rotor, Shared Source CLI Provides Source Code for a FreeBSD Implementation of. NET. MSDN Magazine, 2002
- [Rudalics90] M. Rudalics. Correctness of distributed garbage collection algorithms. Technical Report 90-40.0, Johannes Kepler Universität, Linz, 1990
- [Russo91] Vincent Russo. Garbage collecting an object-oriented operating system kernel. Workshop on GC in Object-Oriented Systems at Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91).

References (S1)

- [Samples92] A. D. Samples. Garbage collection-cooperative C++. Proc. Int. Workshop on Memory Management, 637 in Lecture Notes in Computer Science, pages 315--329, Saint-Malo (France), September 1992. Springer-Verlag.
- [Sánchez01] Alfonso Sánchez, Luís Veiga, Paulo Ferreira: Distributed Garbage Collection for Wide Area Replicated Memory. COOTS 2001: 61-76
- [Shapiro90] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapports de Recherche 1320, INRIA-Rocquencourt, November 1990. Superseded by [Shapiro91]
- [Shapiro91] Marc Shapiro. A fault-tolerant, scalable, low-overhead distributed garbage detection protocol. Tenth Symp. on Reliable Distributed Systems, Pisa (Italy), October 1991.

References (S2)

- [Shapiro92] Marc Shapiro, Peter Dickman, and David Plainfossè. Robust, distributed references and acyclic garbage collection. Symp. on Principles of Distributed Computing, pages 135--146, Vancouver (Canada), August 1992. ACM.
- [SIGPLAN91] Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Ontario, June 1991. ACM Press. Published as SIGPLAN Notices 26(6), June 1992.
- [Skubi97] M. Skubiszewski and P. Valduriez. Concurrent garbage collection in O2. In M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, and M.A. Jeusfeld, editors, VLDB'97 Proceedings of 23rd International Conference on Very Large Databases, pages 356-365, Athens, May 1997. Morgan Kaufman.
- [Steele75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. Communications of the ACM, 18(9):495--508, September 1975.
- [Steenkiste87] Peter Steenkiste and John Hennessy. Tags and type checking in Lisp. Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pages 50--59, Palo Alto, California, October 1987.

References (U-V)

- [Ungar87] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pages 157--167. ACM Press, April 1984. Published as ACM SIGPLAN Notices 19(5), May, 1987.
- [Ungar88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88) Proceedings, pages 1--17, San Diego, California (USA), September 1988. ACM Press.
- [Veiga03] Luís Veiga, Paulo Ferreira: Complete distributed garbage collection: an experience with Rotor. IEE Proceedings - Software 150(5): 283-290 (2003)
- [Veiga05] Luís Veiga, Paulo Ferreira: Asynchronous Complete Distributed Garbage Collection. IEEE 19th International Parallel and Distributed Processing Symposium, Denver, Colorado, USA, April IEEE Computing Society Press, 2005
- [Veiga05a] Luís Veiga and Paulo Ferreira. A Comprehensive Approach for Memory Management of Replicated Objects. INESC-ID Technical Report RT/07/2005, Lisbon, Portugal, April 2005
- [Vestal87] Stephen C. Vestal. Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming. PhD thesis, University of Washington, Seattle, WA, 1987.

References (W-Y-Z)

- [Watson87] P.Watson and I.Watson. An efficient garbage collection scheme for parallel computer architectures. PARLE'87---Parallel Architectures and Languages Europe, number 259 in Lecture Notes in Computer Science, Eindhoven (The Netherlands), June 1987. Springer-Verlag.
- [Weiser89] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. Proceedings of the Twelfth Symposium on Operating Systems Principles, Litchfield Park, AZ (USA), December 1989. ACM Press.
- [Wentworth90] E. P. Wentworth. Pitfalls of conservative garbage collection. Software Practice and Experience, 20(7):719--727, July 1990.
- [Wilson92] Paul R. Wilson and Sheetal V. Kakkad. Pointer swizzling at page fault time: Efficiently and compatibly supporting huge address spaces on standard hardware. International Workshop on Object Orientation in Operating Systems, pages 364--377, Dourdan (France), October 1992.
- [Yuasa90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. Journal of Systems and Software, 11:181--198, 1990.
- [Zorn89] Benjamin Zorn. Comparative Performance Evaluation of Garbage Collection Algorithms. PhD thesis, University of California at Berkeley, EECS Department, December 1989. Technical Report UCB/CSD 89/544.