

A Soft Real-Time Precise Tracing Garbage Collector for C++

Piotr KOŁACZKOWSKI

e-mail: P.Kolaczkowski@elka.pw.edu.pl

Ilona BLUEMKE

email: ibl@ii.pw.edu.pl

Warsaw University of Technology, Institute of Computer Science
ul. Nowowiejska 15/17, 00-665 Warszawa

21.11.2004

Abstract: The paper presents an efficient garbage collector that can be used in soft real-time and interactive applications written in C++. The garbage collector is tuned to best meet requirements of Internet server-side applications. It collects and sweeps garbage concurrently to application threads, thus it minimizes unpredictable execution delays and can increase applications' performance. The article shows how such garbage collector is built, what data structures and algorithms are used, and how they can be efficiently implemented in C++.

Keywords: memory management, mark-and-sweep algorithm, smart pointer, real-time application, concurrent garbage collection, object finalization, write barrier, reference cycles.

1. Introduction

Garbage collection is a mechanism that enables programmers to create software of higher quality than using standard manual memory management schemes. Automatic memory manager would never deallocate the same memory region twice or deallocate memory being used, causing a memory access error or a potential security hole. For applications running during long time without reloading, even a small memory leak can be a disaster. Though it is possible to avoid all these errors without a garbage collector, using it can significantly reduce total software production time. All frameworks for creating enterprise applications (J2EE, .NET) include powerful garbage collectors.

C++ is a widely used high-level programming language not equipped with a standard

garbage collector. A designer can choose a solution best suited for a specific application by installing a proper library. There are many types of garbage collection schemes and none is a „silver bullet”. In table 1 the most commonly used types of garbage collection techniques in C++ programs are shown and also some of their characteristics are given.

Table 1. Comparison of garbage collector techniques in C++

| Garbage collection type | Pros | Cons | Example implementations |
|--|--|---|--|
| Reference counting garbage collectors | <ul style="list-style-type: none"> • Easy to implement in C++ using „smart pointers”. • Reclaims memory just after the last reference to object is dropped. • Applications usually have shorter response times than those using blocking mark-and-sweep collectors. | <ul style="list-style-type: none"> • Perform very slowly, especially in multi-threaded applications. • Can make unpredictable delays. • Cannot reclaim reference cycles. | <ul style="list-style-type: none"> • C++ Boost Library [1] |
| Conservative tracing blocking garbage collectors | <ul style="list-style-type: none"> • Perform fast. • Reclaim reference cycles. • There is no need to rewrite existing code to adapt it to use a garbage collector. • Possible incremental extensions (but slow). | <ul style="list-style-type: none"> • Not precise, very small memory leaks possible. • Must stop all threads for a long time to perform garbage collection. • Often architecture dependent. | <ul style="list-style-type: none"> • Boehm-Demers-Weiser conservative garbage collector [8] |
| Precise tracing blocking garbage collectors | <ul style="list-style-type: none"> • Reclaim reference cycles. • Precise. No leaks. • Architecture independent. | <ul style="list-style-type: none"> • Slower than conservative garbage collectors. • Must stop all threads for a long time to perform garbage collection. | <ul style="list-style-type: none"> • Smieciuch garbage collector by Sebastian Kaliszewski [3] |
| Incremental precise garbage collectors | <ul style="list-style-type: none"> • Can be used in real-time applications - no unpredictable delays. • Reclaim reference cycles. | <ul style="list-style-type: none"> • Usually much slower than blocking collectors. • Higher memory overhead than in the other garbage collector types. | <ul style="list-style-type: none"> • Reverse Trace Garbage Collector by Steve Favor [2] |

| Garbage collection type | Pros | Cons | Example implementations |
|---------------------------------------|--|--|--|
| Concurrent precise garbage collectors | <ul style="list-style-type: none"> • Can be used in real-time applications - no unpredictable delays. • Reclaim reference cycles. • Can run on a multiprocessor machine. • Can be stopped and resumed. • Can use idle application's time. | <ul style="list-style-type: none"> • Difficult to implement. • Usually slightly slower than blocking collectors. | <ul style="list-style-type: none"> • The garbage collector described in this article. |

Unfortunately the C++ language and its compilers are not designed to give any support to garbage collection mechanisms, so programmers must overcome many difficulties [4, 9]. That is probably the reason why other types of garbage collection like compacting, copying or generational garbage collectors [6] are not used (these collectors are implemented in Lisp, Java or .NET). The lack of interactive* or soft real-time** garbage collector C++ makes developing desktop applications, computer games, distributed or web systems more expensive and difficult. The most popular collectors are often not suitable for these applications because:

- A blocking (“stop-the-world”) collector can impose too large unpredictable execution delays. In a simple desktop application a few seconds delay can annoy the user. In a distributed system such delay can lead to a request timeout. In a computer game the delay of 100 ms can distract the player.
- A conservative collector cannot completely avoid memory leaks when a program is running for a long time (example - a web server). This might cause problems in applications that have large heaps.
- An incremental collector works only during a program execution. It does nothing while the program is waiting for user input or a request. Most Internet applications or components in distributed systems spend lots of time simply waiting.

The concurrent garbage collector presented in this article is intended to meet requirements of interactive and soft real-time applications.

2. Basic Concepts

Presented in this paper garbage collector uses *3-colour marking* mark-and-sweep algorithm for concurrently finding inaccessible objects. The algorithm is based on a mark-and-sweep algorithm presented in [9]. It consists of two main phases – a mark phase and a sweep phase. In the mark phase, garbage collector finds the objects that are inaccessible and can be deallocated. In the sweep phase, the destructors of these objects are called and the memory is released.

* Interactive are those applications that should react quickly (in less than a few seconds) to user actions. A word processor is for example an interactive application.

** In soft real-time applications failing to meet a deadline is considered neither application nor system failure, though it is considered as “not good”. An example of a soft real-time application is MPEG-player or 3D computer game where number of displayed frames per second should not drop below some level.

Mark phase

Each application using automated memory management consists of two parts: a *memory manager* and an actual application, called further a *mutator*. *Live object* (potentially accessible object) is a memory block that can be accessed by the *mutator*. The memory manager knows where all the memory blocks are, because it allocates them as a result of mutator's demand. A memory block "X" can be accessed by the mutator if and only if:

- There is a pointer on the application's stack, in the global variable or in the processor's register pointing to the beginning of the memory block "X" or inside it. Such a memory block is called *directly accessible*.
- There is a pointer in memory block "Y" on the heap pointing to the beginning of the memory block "X" or inside it. Then memory block "X" is called *indirectly accessible*.

The garbage collector must know all the pointers pointing to indirectly accessible memory blocks. The set of these blocks is called a *root set*. A basic idea of the mark phase is that starting from the root set, the collector marks each reachable object as live. Then, all unmarked objects are deallocated. This would work fine, if the mutator couldn't change pointer values during the mark phase, as it is in the blocking collectors. If the pointers can be changed in the mark phase, some pointers can be hidden to the collector so it could reclaim live memory. In the concurrent algorithm implemented more sophisticated marking method is used. To better understand it, it is good to imagine, that each memory block can be coloured in one of three colours: white, grey and black.

- *White*: Objects that were not scanned by the collector.
- *Grey*: Objects that were not scanned by the collector, but are known to be used because they are pointed by a pointer caught by a *write barrier*. Only white objects can be greyed. Black objects remain black until marked white.
- *Black*: Objects that were scanned, and are known to be reachable from the root set.

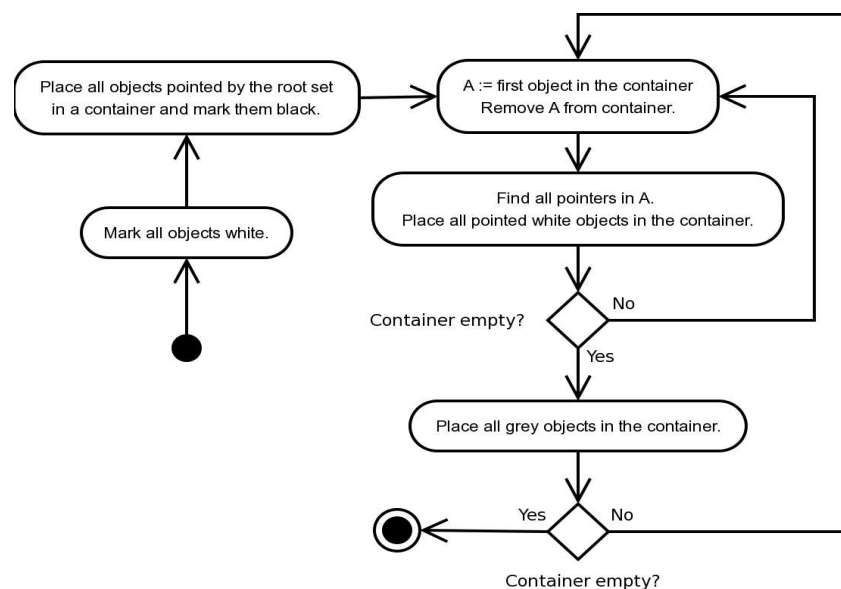


Figure 1 Marking phase

The idea of the colouring algorithm is presented in Fig. 1. Grey objects are used to prevent

the mutator from hiding any pointers during the mark phase. With each write of a pointer, the newly pointed object is marked grey. After the end of the standard mark phase, an additional pass finds all grey objects, uses them as a new root set, and begins the marking procedure once again. The process is repeated until no more grey objects can be found.

Sweep phase

The sweep phase destroys all objects which remain white after the marking phase and consists of two sub-phases. The main activities of the sweep phase are shown in Fig. 2. Firstly, all destructors of unreachable objects are called. Calling an object's destructor and then immediately deallocating the memory of the object, might cause some problem. The other destructors might still reference the already destroyed object. To overcome this problem the memory is reclaimed in the second sub-phase, after the last destructor had been called. In this solution a weakness can still be found. There is no guarantee, that unreachable objects do not have references to the reachable ones. This situation is rather unlikely to happen. Destructors should release system resources and might set some pointers to *null*. Thus, the programmer is responsible for writing a finalization code that doesn't make objects "resuscitate". However the collector is able to detect such objects, keeping them in memory makes it possible to use an object after its finalization. The detection of "resuscitated" object is the collector option and can be chosen in the debug mode. If this option is on, the collector terminates the application and shows the suspected destructors' address to the programmer. After some changes in the code of the destructor, the programmer can turn the option off. Detailed aspects of writing destructors and finalizers are given in [5].

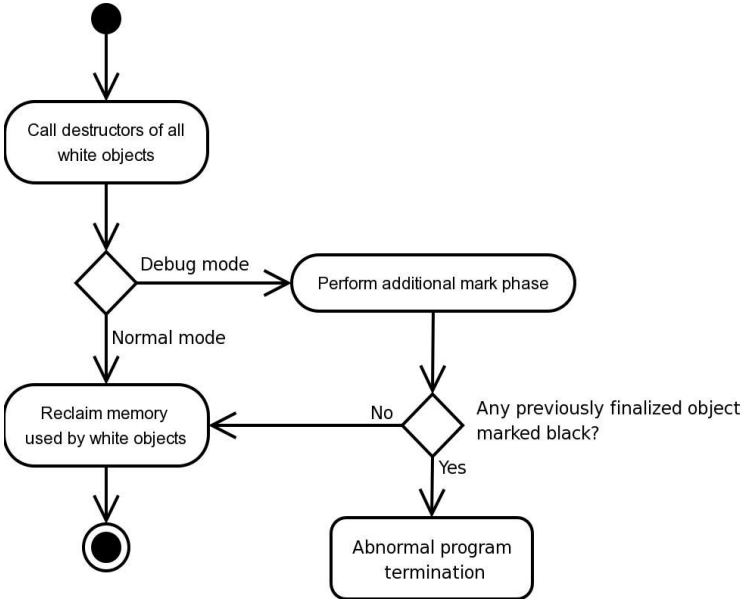


Figure 2 The sweep phase

3. Data Structures and Algorithms

The garbage collector stores information about pointers, objects and their types in its internal data structures. These structures should enable the collector easy and fast access. Some parts of data should also be accessible by the mutator, so the synchronisation aspects must be

taken into account.

Pointers

Because of lack of support for *runtime type information* in most C++ compilers, it is not possible to use built-in C++ pointers with a precise collector. It could not distinguish pointers from variables of other data types. In this collector a *smart pointer* design pattern has been used [7, 9]. C++ templates and operator overloading are used to emulate the standard C++ pointer behaviour. The pointer template consists of two fields:

- `ObjectPtr` – Points to the user's C++ object. This field has the same value as the standard C++ pointer.
- `BlockPtr` – Points to the beginning of the continuous memory block allocated for the object. This memory block contains the object and additionally, some internal collector's data describing the object. This internal data is called a *block descriptor*. The block descriptor contains for example a *colour field* which allows colouring the block in one of the three colours as described in section 2.

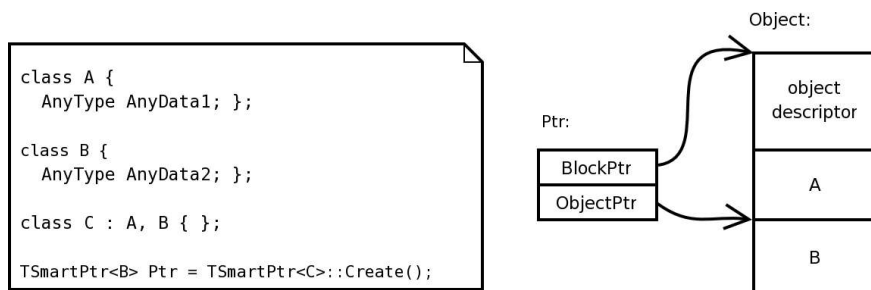


Figure 3: A pointer to an object - a C++ definition and its representation in memory.

One single field (the second one) is insufficient, due to required support for multiple inheritance of classes. To allow class inheritance, a C++ compiler sometimes changes a pointer value when it is casted into a compatible pointer type. `BlockPtr` cannot be changed. For this purpose `ObjectPtr` is used. On the other hand the exact location of an object's descriptor is required for the mark phase to mark objects and get their pointers, so `ObjectPtr` alone would be insufficient. Example pointer definition with its memory representation is shown in Fig. 3.

There are two types of pointers: direct and indirect. *Direct pointers* can be accessed directly from processor's registers, application stack, global variable or manually allocated memory block on the heap. *Indirect pointers* are accessed indirectly from an object managed by the collector. The direct pointers point to the objects used as a root set in the beginning of the mark-phase. Addresses of these pointers build a *pointer root set* which is kept in a hash table. The pointer's constructor decides if the pointer is direct or indirect and places all the direct pointers in the container. The destructor is responsible for removing direct pointers from the hash table. There is a good criterion for checking if a pointer is indirect. Every indirect pointer lies in a memory block created by the garbage collector. Just after creating a memory block the collector places its address and size on the special stack called *new objects stack*. Then the constructor of object is called, and if it contains any members of pointer type, it calls their constructors too. Each pointer constructor checks, if a pointer's address is positioned inside the block which position is stored on the top of the new objects stack. If not, the pointer is added to

the pointer root set. When all constructors are called, the position of object is taken from the stack. Object addresses are kept on the stack because it is possible, that a constructor of object creates some more objects.

Objects

As mentioned above, each memory block allocated by the collector, starts with an appropriate object descriptor. The descriptor consists of:

- “Is object black” boolean flag.
- “Is object grey” boolean flag.
- Index of the *type descriptor* in the *type descriptor array*. Type descriptor contains information about relative locations of pointers in all objects of this type.

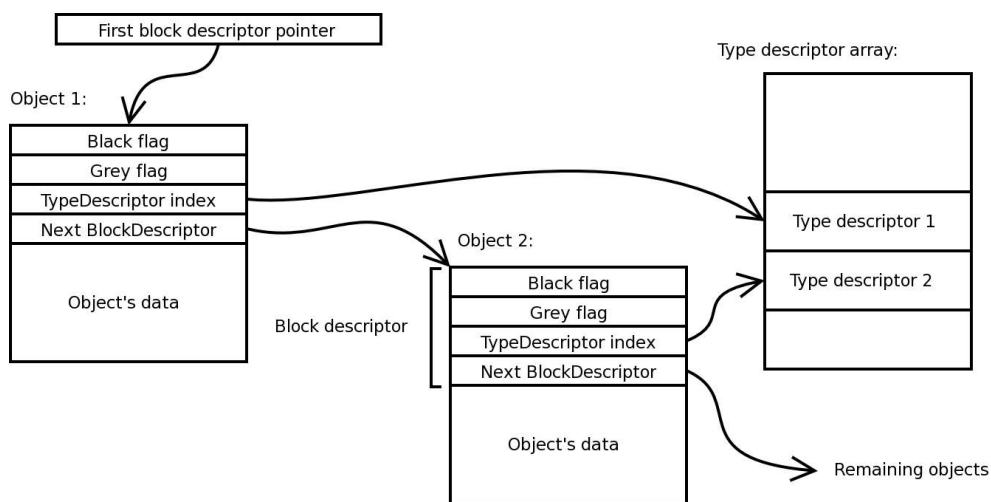


Figure 4: Objects of different type.

- Pointer to a next object descriptor. These pointers are used to form a single-linked list of all allocated blocks. This list is managed by the collector. It stores a pointer to the first and last allocated memory block.

The list of objects (as shown in Fig. 4) is easy to use, and, what is very important, an operation of adding and removing element has a constant complexity. It would be possible to store object descriptors in a separate vector, but such vector would sometimes require resizing. Resizing the vector can lead to reallocating its memory and could stop the mutator, for the time proportional to the number of all allocated objects. The pause of the mutator can be unacceptable in a real-time application. Objects are created by any mutator's thread and destroyed by the collector's thread in the sweep phase. In the list of objects new objects are created always at the end of the list. This solution has other advantage. It is possible to avoid synchronization of collector and mutator threads for all objects except the last one. Only one mutator's thread can create objects at a time.

Runtime type information

Runtime type information are kept in the type descriptors which are used by the collector to find all pointers and read their values. Each type descriptor stores:

- Object's size.
- A list of positions of the pointers in the object relative to the beginning of the memory block in which the object is allocated.
- A pointer to the object's destructor.
- A complete/incomplete flag. This flag is used when the type descriptor is built for the first created object of that type.

Type descriptors are kept in an array which can be resized. Each type descriptor is created with a first instance of an object. Object creation is not concurrent, so only one type descriptor can be written at a time. A list of relative positions of pointers is created by a pointer constructor. When the constructor finds out, that the pointer is indirect, it checks if the type descriptor of the newly created object is complete (it can access the type descriptor because a reference to it is placed also on the new object stack). If not, it adds the difference between the address of the pointer and the address of the object (again uses the new object stack) to the list of relative positions of pointers in the type descriptor. After the instance of object has been created, the type descriptor is marked complete.

Triggering the collection

A very important in a garbage collector is to have a good algorithm deciding when to collect garbage. If the collector is triggered too often, then the application uses less memory, but performs slowly. Each collection requires touching all live memory so it makes processor's cache less efficient. If the collector is triggered too rarely, the application uses much more memory than it actually needs, but the cache performs better. In some systems, the garbage collector collects garbage only when there is not enough memory to allocate a new object. This is a bad idea for a real time garbage collector, because it can stop a mutator for the whole collection cycle. In present operating systems it is also difficult to tell when the application is lacking memory, because the OS can swap some pages out and the whole “visible” memory to the application is much larger than physical memory in the system. Delaying collection until the system starts to swap would lower significantly the system's performance. This garbage collector starts collecting garbage, when one of the following conditions appears:

- Total memory amount allocated on the heap by the collector exceeds a limit defined by the application programmer.
- Memory amount allocated after the last collection exceeds a limit obtained by multiplying some constant factor by the total memory amount allocated on the heap. This constant factor can be tuned by the application programmer. This makes period between collections proportional to the amount of allocated memory, unused memory amount proportional to the used memory amount and keeps the collector's load at a constant level.
- The programmer manually triggers the collection for example when he/she knows that the application produces too much garbage or it will be idle for a long time.

4. Performance

Like every garbage collector, the collector presented in the paper incurs some overheads:

1. Each write of a pointer requires additional greying of an object (*write barrier*).
2. Each copying of a pointer requires copying doubled amount of memory then when copying a standard C++ pointer.
3. Each pointer creation and destruction requires checking, if it is direct or indirect, and

eventually places it in the hash table. This operation must be protected by critical section and is described in more details below.

4. Each object creation requires placing and removing some special information on/from the new objects stack.
5. The main garbage collector's thread that collects and sweeps garbage requires some time proportional to the heap size.

The most severe overhead is mentioned in the third point. Entering a critical section takes more time than placing one simple value into the hash table. A good solution is to place a queue between the mutator's thread that creates pointers and the collector. Pointer's constructor puts the pointers address into this queue and when the queue is full, it enters a critical section and copies all data from the queue into the hash table. The removal of pointers' addresses from the hash table is made in the same way. This solution enables the mutator entering the critical section only once per hundreds of pointer creations. The queues should be smaller than the processor's cache. Searching reachable and non-reachable objects incurs almost no overhead in this type of collector if the mutator is idle for some time. The collector can use this idle time to collect garbage.

In Table 2 some results of tests executed on Intel Celeron 2.4 GHz, 128 kB L2 cache, 256 MB total RAM, OS: Linux kernel 2.4.25 for two memory management schemas are presented.

Table 2: Execution times using two different memory management schemes.

| | Execution times [ns] | |
|---------------------------------------|--------------------------|-----------------------------|
| | Manual memory management | Automatic memory management |
| Dereferencing a pointer | 1.0 ± 0.1 | 1.0 ± 0.1 |
| Writing a pointer | 1.0 ± 0.1 | 2.2 ± 0.2 |
| Creating and destroying a pointer | 0.0 | $\leq 56 \pm 6$ |
| Creating and destroying a 24 B object | 360 ± 40 | $\leq 1100 \pm 150$ |

The responsiveness of programs using presented garbage collector can be better than if they used blocking garbage collector. There are only few cases, when mutator's thread can be stopped by the collector:

1. A thread is allocating a new object. Allocation of objects is not concurrent, so if any other thread also wants to create a new object, one of the threads have to wait for the other one. Thus the maximum delay time is proportional to the number of threads. The only thing that can unpredictable delay allocation is the system's allocator. The collector can delay allocating a new object by a constant amount of time if the sweep phase is currently removing the last object in the object list.
2. A thread is deregistering a pointer kept in the pointer root set and the garbage collector is scanning a root set. A thread is suspended till the collector finishes with the root set. This delay is proportional to the number of directly accessible objects. This number remains usually much smaller than the total number of objects and can be easily predicted at the time the program is designed. Unless program uses deep recursive calls that allocate pointers at each recursion level, the number of root pointers is limited.
3. A thread is registering a pointer and the queue storing pointers to register is full. If the

collector is scanning the root set, the thread has to wait for it to finish scanning. Otherwise, the threads waits only during the writing of all pointers from queue into the hash table. The queue has a constant size.

It can be noticed, that a thread that does not work with smart pointers is never suspended by the garbage collector. The thread can use as much CPU time as it gets from the operating system. Of course, the operating system's thread switching policy influences the application's response times.

5. Summary and Future Work

This paper presents an implementation of an garbage collector written entirely in standard C++ and using smart pointers. The collector has following features:

- It can be used with any C++ program in environment that supports multithreading.
- It doesn't influence any code that manages memory manually. Some objects can be managed manually and some automatically in the same program.
- Execution of a process is interrupted for a predictable, limited in most cases, amount of time.
- Idle time of the process or another CPU can be used for collecting garbage, so the collector should be perfect for programs with “request-reply” execution model.

Future works will focus on:

- Reducing a total object creation time and heap fragmentation by using a dedicated memory allocator. Such allocator, to perform better, can use data stored in the collector.
- Reducing a total pointer creation time by treating root pointers placed on the stack different than root pointers placed on the heap. The majority of pointers are those placed on the stack.
- Improving reference locality by introducing a generational mark-and-sweep algorithm. A slight complication of the write barrier will be required, so it is not sure that this change will really improve the performance of garbage collector.

Bibliography

- [1]: http://www.boost.org/libs/smart_ptr/smart_ptr.htm
- [2]: <http://sourceforge.net/projects/librtgc/>
- [3]: <http://smieciuch.sourceforge.net/>
- [4]: A. Divan, E. Moss, R. Hudson, Compiler Support for Garbage Collection in a Statically Typed Language, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992
- [5]: Boehm H. J., Destructors, Finalizers, and Synchronization, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Programming Languages*, 2003
- [6]: D. Doligez, X. Leroy, A concurrent, generational garbage collector for a multithreaded implementation of ML, *Proc. 20th symp. Principles of Programming Languages*, 1993
- [7]: Edelson D. R., Pohl I., They're Smart, but They're Not Pointers, *Proceedings of the 1991 Usenix C++ Conference*, 1991
- [8]: J.H.Boehm, Space Efficient Conservative Garbage Collection, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993
- [9]: R. P. Wilson, Uniprocessor Garbage Collection Techniques, *Proceedings of the International Workshop on Memory Management*, 1992