

Piotr Kołaczkowski

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

A Soft Real-Time Precise Garbage Collector for Multimedia Applications

Abstract: The paper presents an efficient garbage collector that can be used in soft real-time and interactive multimedia applications written in C++. The garbage collector collects and sweeps garbage concurrently to application threads, thus it strongly reduces risk of memory leaks and it minimizes unpredictable execution delays. This feature makes it usable for many resource-demanding applications like computer games, media players, multimedia plugins in web-browsers, digital encyclopaedia or web-based systems. 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, forget to release unused memory or deallocate memory being used causing a memory access error. In applications running for a long period of time without reloading and applications processing multimedia content requiring lots of memory, even a small memory leak can be a disaster. Though it is possible to avoid all these errors without having a garbage collector, using it can significantly reduce total software production time.

C++ - widely used high-level programming language is not equipped with a standard garbage collector. A designer can choose a solution best suited for a specific application by installing a proper library. Unfortunately the C++ language and its compilers are not designed to give any support to garbage collection mechanisms, so programmers must overcome many difficulties [2, 5]. That is probably the reason why other types of garbage collection like compacting, copying or generational garbage collectors [3] 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 various types of multimedia applications like web-browsers, media players or computer games more expensive and difficult. The most popular collectors are often not suitable for these applications because:

* 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 the number of displayed frames per second should not drop below some level.

- 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 computer game a 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 (e.g. 3 hours long video), especially if it uses lots of memory. Memory leaks can lead to abnormal program termination. This is not acceptable in most applications.
- An incremental collector works only during a program execution. It does nothing while the program is waiting for user input or a request. Interactive applications spend most of their time simply waiting.

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

2. Algorithms and data structures

Presented garbage collector uses a smart-pointer design pattern for determining exact positions of pointers in the application's memory [4, 5] and *3-colour marking* mark-and-sweep algorithm for finding inaccessible objects. The algorithm is based on a mark-and-sweep algorithm presented in [5]. 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.

2.1. 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.

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.

2.2. Sweep phase

The sweep phase destroys all objects which remain white after the marking phase and consists of two sub-phases. 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. Detailed aspects of writing destructors and finalizers are given in [1].

2.3. 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. 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.

3. 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 than 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' addresss 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. Two memory management schemes are presented. Total object destruction times would be lower if application remained idle for some time.

Table 1: 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	$\leq 40 \pm 5$
Creating and destroying a 24 B object	360 ± 40	$\leq 1100 \pm 150$

This micro-benchmark results shows that execution time overheads would be very high in programs creating many temporary objects with corresponding pointers. Fortunately in C++ most temporary objects are placed on the stack, so they do not need garbage collection at all. The garbage collector is used mostly for long living, usually large objects. Table 2 shows performance of the garbage collector in a “real-world” application like web-server.

Table 2: Execution times of a simple web-server serving multimedia content.

	Execution times [s]	
	Manual memory management	Automatic memory management
10000 requests, each of size 10 kB	48 ± 3	50 ± 3

The predictability of performance of programs using presented garbage collector can be much 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. Table 3 shows response times of the simple web server sequentially responding to 10000 requests. Both client and server were run on the same machine.

Table 3: Response times of a simple web-server serving multimedia content.

	Response times [ms]
Maximal:	16 ± 5
Average:	4.8 ± 0.7
Minimal:	3.1 ± 0.5

4. Summary and Future Work

This paper presents an implementation of a garbage collector written entirely in standard C++ and using smart pointers. The collector is suitable for most multimedia applications and has following features:

- Execution of a process is interrupted for a predictable, limited in most cases, amount of time. The effective maximal/average/minimal response times depend only on the allocator and the scheduling algorithm of the operating system's kernel. This feature makes the garbage collector particularly useful in sound/video real-time processing or computer games.
- 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.
- Idle time of the process or another CPU can be used for collecting garbage, so the collector should be perfect for interactive programs.
 - All inaccessible objects are collected, so no memory leaks occur even in long running programs.

It has been also shown that using this garbage collector doesn't need to have any significant influence on application's performance or response times. However, further tests must be done to confirm this statement.

Future works will focus on:

- Further performance improvement by applying more sophisticated techniques like generational garbage collection.
- Testing the garbage collector in various multimedia applications.

Dokładny odśmieczacz czasu rzeczywistego dla aplikacji multimedialnych

Streszczenie: W pracy zaprezentowano wydajny odśmieczacz do zastosowań w aplikacjach multimedialnych czasu rzeczywistego pisanych w języku C++. Odśmieczacz wykrywa i usuwa niedostępne obiekty nie zatrzymując wątków aplikacji, dzięki czemu znacznie ogranicza możliwość wystąpienia przecieków pamięci i nie wprowadza przy tym znaczących opóźnień wykonania programu. Te cechy powodują, że szczególnie nadaje się do zastosowań w aplikacjach wymagających dużej ilości zasobów takich jak: gry komputerowe, odtwarzacze dźwięku/obrazu, wtyczki multimedialne w przeglądarkach WWW, encyklopedie multimedialne czy całe systemy oparte na technologii WWW. Atrykuł przedstawia struktury danych i algorytmy użyte do budowy odśmieczacza, sposób ich efektywnej implementacji w języku C++ oraz ocenę przydatności odśmieczacza w zastosowaniach multimedialnych.

Słowa kluczowe: zarządzanie pamięcią, algorytm "mark-and-sweep", inteligentny wskaźnik, aplikacja czasu rzeczywistego, odśmiecanie współbieżne, finalizacja obiektów, bariera zapisu, cykle odwołań.

Bibliography

- [1]: Boehm H. J., Destructors, Finalizers, and Synchronization, *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Programming Languages*, 2003
- [2]: Divan A., Moss E., Hudson R., Compiler Support for Garbage Collection in a Statically Typed Language, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992
- [3]: Doligez D., Leroy X., A concurrent, generational garbage collector for a multithreaded implementation of ML, *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1993
- [4]: Edelson D. R., Pohl I., They're Smart, but They're Not Pointers, *Proceedings of the 1991 Usenix C++ Conference*, 1991
- [5]: Wilson R. P., Uniprocessor Garbage Collection Techniques, *Proceedings of the International Workshop on Memory Management*, 1992