A Customisable Memory Management Framework for C++ [*]

Giuseppe Attardi, Tito Flagella and Pietro Iglio
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, I-56125 Pisa, Italy
email: `attardi@di.unipi.it, tito@link.it, iglio@fub.it`

Keywords: memory management, garbage collection, C++

Abstract

Automatic garbage collection relieves programmers from the burden of managing memory themselves and several techniques have been developed that make garbage collection feasible in many situations, including real time applications or within traditional programming languages. However optimal performance cannot always be achieved by a uniform general purpose solution. Sometimes an algorithm exhibits a predictable pattern of memory usage that could be better handled specifically, delaying as much as possible the intervention of the general purpose collector. This leads to the requirement for algorithm specific customisation of the collector strategies. We present a dynamic memory management framework which can be customised to the needs of an algorithm, while preserving the convenience of automatic collection in the normal case. The Customisable Memory Manager (CMM) organises memory in multiple heaps. Each heap is an instance of a C++ class which abstracts and encapsulates a particular storage discipline. The default heap for collectable objects uses the technique of mostly copying garbage collection, providing good performance and memory compaction. Customisation of the collector is achieved exploiting object orientation by defining specialised versions of the collector methods for each heap class. The object oriented interface to the collector enables coexistence and coordination among the various collectors as well as integration with traditional code unaware of garbage collection. The CMM is implemented in C++ without any special support in the language or the compiler. The techniques used in the CMM are general enough to be applicable also to other languages. The performance of the CMM is analysed and compared to other conservative collectors for C/C++ in various configurations.

1 Introduction

In most programming languages, memory allocation is either under total responsibility of the programmer or under full control of a garbage collector.

The garbage collector's function is to find data objects that are no longer in use and to reclaim their space for further use by the program. An object is considered *garbage*, and therefore subject to reclamation, if it is not reachable by the program via any path of pointer traversal. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never follow a "dangling pointer" leading to a deallocated object.

This technique has several advantages since it improves: *safety*, avoiding the risk of deallocating an object too soon; *accuracy*, avoiding the risk of forgetting to deallocate unused memory; *simplicity*, assuming a computational model with unlimited memory; *modularity*, the program does not have to be interspersed with bookkeeping code not related to the application; *burden* on programmers who are relieved from taking care of memory management.

[*]The research described here has been funded in part by the ESPRIT Basic Research Action, project 6846, PoSSo: Polynomial System Solving and by ESPRIT Reactive LTR, project 21.024, FRISCO: A Framework for Integrated Symbolic Numeric Computing.

Garbage collection has been mostly available in programming languages whose design had taken into account its requirements: for instance not allowing direct pointer manipulation (Lisp, Java [1]), using tagged pointers or providing run-time type information, using special notation for pointer operations (Simula), requiring enhanced pointer declarations (Modula3).

This has restricted garbage collection from general use, more often than the concerns about efficiency. Recent research has proved in fact that the performance of garbage collectors compares quite well with explicit memory deallocation (using primitives like `free()` or `delete()`) [18], and techniques like *incremental* garbage collection have been developed to reduce latency during garbage collection.

With the development of techniques for *conservative* garbage collection, the use of garbage collection has become feasible also for languages that are not well behaved with respect to pointers. Even so, current implementations limit the use of pointers: for instance in Modula3 [31] there are traced pointers to collected objects and untraced pointers to uncollected objects – an uncollected object can't contain a traced pointer.

A garbage collector in general assumes full control of memory management, and this can be sometimes a drawback. For instance, it may be hard to integrate code or libraries that are unaware of garbage collection and use pointers without restrictions, it may be impossible to mix code from programming languages with different collectors, like for instance Lisp and Prolog. And finally, as argued here, one cannot specialise the collector to the particular needs of an algorithm, except in the form of hints, which some collectors accept.

A general purpose collector strategy may work well in most circumstances, but there are cases where an algorithm within an application exhibits a predictable pattern of memory usage that can be exploited to achieve significant performance benefits. When the collector is alone in control of memory management, it becomes impossible to arrange for allocating and deallocating objects in a special way during the execution of that portion of the application. The programmer could request an area of memory from the operating system and arrange for managing it by himself. This however would not be sufficient if pointers are allowed from within such an area to objects external to it. Such objects might still be reachable but the general collector would not be aware of them and might unduly reclaim their space. Therefore if the user wants to manage memory by himself, some form of coordination with the general collector is necessary: for instance one can inform a conservative collector that such area needs to be searched for pointers.

A situation like this arose in developing memory management facilities for an european research project in symbolic algebra: the ESPRIT BRA PoSSo which developed a sophisticated system for solving systems of polynomial equations. The core algorithm of PoSSo is quite memory intensive and its implementation with traditional garbage collection techniques often resulted in thrashing where most of the time was spent in garbage collection. However, there are precise points in the algorithm where all data created during the previous step of the algorithm become irrelevant and can be deallocated as a block. By customising the allocation within this portion of the algorithm significant improvements in performance have been achieved.

The requirements of this project led us to design a Customisable Memory Management (CMM) framework. In CMM several policies can coexist: users can choose among a few pre-built variants, ranging from manual management to fully automatic garbage collection, but can also implement their own specialised memory management. The extensibility of the framework is achieved exploiting the object oriented paradigm of C++, thereby maintaining a consistent and simple interface for programmers.

The CMM framework consists of two abstract classes:

1. `CmmHeap`: base class for heap implementations

2. `CmmObject`: base class for collected objects

Class `CmmHeap` encapsulates the mechanisms for allocating and scavenging objects within an area of memory called "heap". CMM provides a set of facilities for implementing customised memory management policies.

Along with the abstract class `CmmHeap`, CMM includes a few predefined concrete classes, derived from it, that implement some common policies:

- class `DefaultHeap` provides a general purpose garbage collector which implements a variant of Bartlett's mostly copying collector strategy [8];

- class `TempHeap` is useful when a phase of a program allocates large amounts of temporary objects that can be discarded at the end of the phase;
- class `MarkAndSweep` implements a fully conservative mark-and-sweep collector, which does not require information about objects layout as the `DefaultHeap`, but does not perform compaction.
- class `UncollectedHeap` provides the traditional manual allocation discipline.

The abstract class `CmmObject` provides support for classes of collected objects that must be traced or swept by a collector.

2 Custom Object Allocation

Among the properties of the storage for an object that one would like to be able to exploit or control are:

- lifetime
- relocatability
- traceability

For instance, if an object contains some implicit pointers to within itself, like a branch to subroutine instruction in a binary code segment, one needs to specify that the object cannot be relocated. Supplying information about the layout of an object may be useful to improve the accuracy and the performance of the garbage collector. For instance, an array of characters need not be traversed looking for pointers to other objects. Specifying the lifetime of an object is more difficult, but there are some useful simple cases: for instance asserting that an object is permanent may be useful to store it in an area that is visited less frequently by the collector; objects whose extent is the same as procedures can be allocated on the stack.

Controlling the storage properties of objects may require some careful coding work and should be applied only to well-understood code to avoid the risks of introducing "memory leaks", but nevertheless some applications may demand this for performance reasons.

One simple idea is to exploit the phase behavior of programs, by segregating in different "zones" of memory those objects which do not survive a phase. This approach dates as far back as Collins' system [15] and can be used even without a garbage collector. Hanson [25] presents such technique as used in the GNU C compiler system: a special kind of heap is created on demand, called an "obstack", short for "object stack". Objects known to die at the end of a phase can be allocated on an obstack and all freed at once when the phase is over. This is often more efficient and convenient than traversing data structures being deallocated and freeing each object individually. However, the solution will not be effective if the amount of storage required within a phase is not limited. We will show later how the idea of recovering memory in block can be exploited in our CMM system in conjunction with a collector that may be triggered within a phase. Further discussion of Hanson's technique and a detailed and informative survey on other techniques for conventional allocation is presented in [44].

Multiple heaps are supported by the Win32 API [34], which provides `HeapCreate()` and `HeapDestroy()` for creating and destroying heaps, `HeapAlloc()` for allocating a block of memory and `HeapFree()` for freeing it.

When a garbage collector is involved, storage properties are usually dictated by the collector design and users have no control on them: if the collector is a copying collector, it will not handle objects that cannot be moved. The lifetime of dynamically allocated objects is unpredicatable and uncontrollable.

To provide user control over these properties entails adding a new dimension to garbage collection: customisation of object allocation, applicable to individual objects rather than to the whole collector or to classes of objects. Customisation requires a collector designed to be open and to delegate portions of its task to other collectors. This is a different concept from the mechanisms of parametrisation or tuning that some collectors provide.

For instance garbage collection intervention can be sometimes avoided in ADA [27] by specifying an upper bound for the space needed for data of a certain type. The corresponding space can then be reserved globally when the definition is elaborated. Subsequently, when leaving the program unit enclosing the type definition, the space corresponding to the collection may be recovered since the contained objects are no longer accessible.

An interesting form of tuning is provided in Lisp Machine Lisp [39] where one can define areas of memory and designate which area to use when allocating objects. Areas are primarily used to give the user control over the paging behaviour of a program. One area could be selected as permanent, so that it would not be copied at each activation of the ephemeral collector [30]. Microcode support was present in the MIT Lisp Machines so that each area could potentially have a different storage discipline, but apparently such feature was never exploited. It is interesting to note that before the ephemeral collector was developed, areas were used to implement an obstack-like mechanism. An heavy use of such mechanism however became a continuing source of storage leaks and its use was discouraged [6].

Information about traversal of objects can be supplied to the Böhm-Weiser collector for C [10] in the form of a region parameter to the allocation routine. Region identification is used to determine how to locate pointers within objects during traversal by the collector. The `PTRFREE` region for instance is used to allocate objects which do not contain pointers. Such regions are simply skipped by the collector. Detailed traversal information for each type of object is instead required in Bartlett's mostly-copying collector [8].

In all these examples however the collector implements a fixed policy, and no alternative is contemplated. The collector routines at most take into account the area where an object resides besides its type and layout.

The CMM allows users to customise object allocation by specifying individually for each object created which policy to adopt for its storage. The policy used for the object is determined by the heap where the object resides, but an object in one heap may get involved in the collection for another heap.

The CMM admits the presence of several collectors, each one in charge of its own heap and establishes conventions for proper overall memory management.

CMM users can select among a few predefined memory management disciplines, define their own, or customise those provided in the framework exploiting the mechanisms of inheritance and specialisation.

For instance a situation like in the following figure is conceivable, where three different memory management policies are available or even used together in the same application: a traditional *stop and copy* collector, a specialised *obstack* allocator for portions of the algorithm with controlled behaviour and a *generational* or *incremental* collector for interactive tasks such as user interfaces.

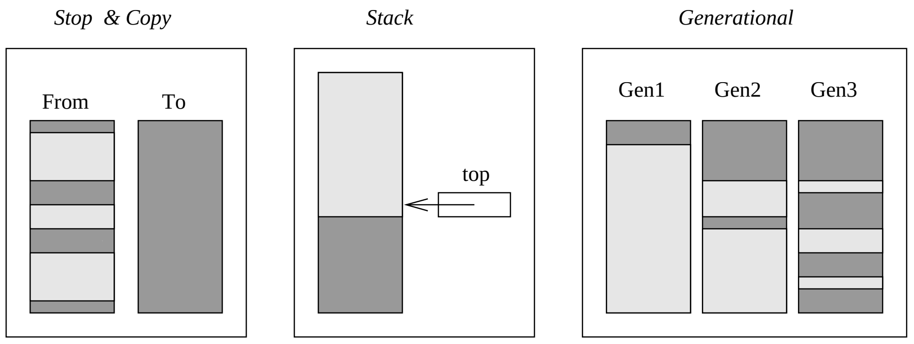

Figure 1: Heaps with different policies

The mechanism to implement alternative policies is the *heap* abstraction which we present later. The generic interface for a heap consists in the typical functions for allocating memory and collecting (`alloc()`, `reclaim()` and `collect()`) plus a function `scavenge()` which is the operation performed by the collector to preserve live objects. The policy for each heap is determined by the algorithms and data structures used in its implementation of these functions.

One could argue whether a single general strategy could fit all the needs. For instance a generational garbage collector ensures that memory is reclaimed quickly. However not even a generational garbage collection is good enough for applications like PoSSo where one must prevent or delay garbage collection as much as possible, not just make its duration shorter. For the majority of applications a general purpose strategy is adequate, and the CMM provides a good one by default.

CMM is meant to provide users with more control over memory allocation than most typical garbage collectors, giving them an alternative between the two extremes (automatic collection vs. manual allocation). Exploiting CMM programmers can develop solutions that in specific situations are more efficient than traditional collectors and are safer than manual allocation, provided of course that their design is careful and is applied in the appropriate cases.

Another problem with relying on a single technique is that it may be difficult to determine when that one technique is not performing well: therefore being able to compare different techniques can be useful to highlight strengths and weaknesses of each technique. This indeed occurred to us in the experiments we report later, when the comparison among different collectors showed a significant difference in certain cases. Without a point of reference we would not have been led to investigate the situation and to discover the reason why the technique was performing poorly and to remedy. A single framework where several techniques are available makes such comparisons simpler to perform.

3 Design Goals

In designing the CMM we tried to achieve the following goals:

- *portability*: the CMM is simply a library of C procedures and C++ classes, which can be used with any C++ compiler and does not rely on changes to the underlying language or compiler.
- *coexistence*: code and objects built with the CMM can be exchanged with traditional code and libraries. No restrictions exist on whether a collected object can point to a non collected object and vice versa. We wanted to be able to pass collected objects to programs unaware of garbage collection, allowing them to store such objects in data structures, without special burden on the programmer or risk that the object would be garbage collected. Alternative solutions require the programmer to put an object in an "escape list" before passing it to an external procedure.
- *algorithm specific customisation*: the allocation policy can be customised to the particular needs of an algorithm. This is different from other solutions, where the allocation policy is associated to the type of an object [22]. For the purpose of our applications, it is necessary to allocate the same type of object sometimes with one policy and sometimes with another. For example, in the PoSSo algebra system there is only one class of polynomials, but a polynomial is allocated in an obstack-like heap when its lifetime is limited within one simplification step of the algorithm and in the general heap when its lifetime cannot be predicted.
- *multiple logical heaps*: one heap is used for collectable objects and one for traditional objects. Other special heaps can be created to accomodate the requirements for special allocation disciplines.
- *usability*: a minimal burden is placed on the programmer who wants to use the collector. To define collectable objects the programmer needs just to specify inheritance from the base class `CmmObject`. Additionally a method for traversing these objects must be supplied if the copying collector is used. This task could be automated.
- *separation of concerns*: memory management code needs not to be included in operations on objects, and the memory policy can be changed just by selecting which heap to employ in an algorithm.
- *efficiency*: the implementation is efficient enough to be as good as and sometimes better than hand tuned allocation.

4 Dynamic Memory Management: Concepts and Terminology

A garbage collection mechanism basically consists of two parts [43]:

1. distinguishing the *live objects* from the garbage in some way, or *garbage detection*;

2. reclaiming the garbage objects' storage, so that the running program can reuse it.

The formal *criterion* to identify *live* objects is expressed in terms of a *root set* and *reachability* from these roots. The *root set* consists of the global and local variables, and any registers used by active procedures. Heap objects directly reachable from any of these variables can potentially be accessed by the running program, so they are *live* objects that must be preserved. In addition, since the program might traverse pointers from these objects to reach other objects, any object reachable from a live object is also live. Thus the *live set* is the set of objects in some way reachable from the roots. Any object not in the live set is garbage and can be safely reclaimed.

Depending on the kind of information available during the traversal of objects from the root set, a tracing collector can be *conservative pointer finding*, *type-accurate* or both.

A *conservative* (*pointer finding*) garbage collector does not require cooperation from the compiler and assumes that anything that *might* be a pointer actually *is* a pointer. In this case an integer (or any other value) is assumed to be a pointer by the collector if it corresponds to an address inside the current heap range: any such value is called an *ambiguous pointer*. A root containing an ambiguous pointer is called an *ambiguous root*. A garbage collector is *type-accurate* when it is able to distinguish which values are genuine pointers to objects.

The main limitations of a purely conservative collector are memory fragmentation in applications dealing with objects of several sizes, which arises from the inability to move objects, and the risk that a significant amount of memory might not be reclaimed in applications with densely populated address spaces of strongly connected objects [41]. Böhm [11] discusses the sources of excess retention and proposes techniques to limit it.

The alternative approach, which is *type-accurate* in identifying objects, faces some problems with hidden pointers. For instance in C++ the location on the stack of the pointer to the object itself, denoted by the variable `this`, is only known to the compiler. The only compiler-independent way to catch such pointers is to examine the stack conservatively. Failing to trace hidden pointers may lead to *dangling pointers* and produce serious consequences for the integrity of the program.

In Bartlett's *mostly copying garbage collector* [7] a partially conservative approach is used: ambiguous roots are dealt conservatively while objects in the heap are dealt in a type-accurate way.

The CMM provides both a mostly-copying collector and a fully conservative mark-and-sweep collector.

The CMM can deal with multiple heaps, each one with a different allocation and collection policy. When allocation in one heap fails, the collector for that heap is normally triggered. Such collector typically reclaims memory only from that heap, even though other heaps may be involved in the process as explained in the following sections. This allows designing specialised heaps that maintain enough information to perform a collection limited to their own area without necessarily involving a global collection.

We start with the description of the collector for the default CMM heap.

5 The Default Heap

The CMM relies on an underlying general mechanism for identifying objects, moving them and recovering memory. The *default collector* of CMM uses these mechanisms to implement Bartlett's technique [7]. The difference and the derivation of our technique from Bartlett's original are discussed in [2]. Here we present our implementation.

A mostly-copying garbage collector performs compacting collection in the presence of ambiguous pointers in the root set. The technique is an evolution of the classical stop-and-copy collector that combines copying and *conservative* collection.

The heap used by the mostly-copying collector consists of a number of equal size pages, each with its own *space-identifier* (either *From* or *To* in the simplest non generational version). The *FromSpace* consists of all pages whose identifier is *From*, and similarly for the *ToSpace*. The collector conservatively scans the stack and global variables looking for potential pointers. Objects referenced by ambiguous roots are not copied, while other live objects are copied. If an object is referenced from a root, it must be scavenged to survive collection.

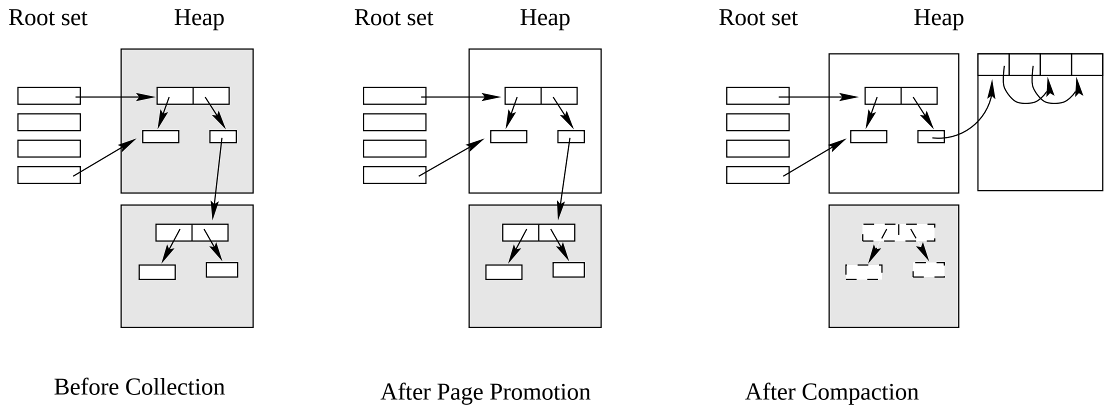

Figure 2: Mostly-copying collector

If the root is ambiguous, the object cannot be moved since the location of the root cannot be updated to the new position, therefore the whole page to which it belongs is preserved. This is done by *promoting* the page into *ToSpace* by simply changing its page space-identifier to *To*. At the end of this promoting phase, all live objects residing in pages of *FromSpace* can be copied and compacted into pages belonging to *ToSpace*. Root reachable objects are traversed with the help of information provided by the application programmer: the programmer must define the member function `traverse()` for each class of objects, to trace the internal pointers within objects of that class.

Our algorithm uses a bit table called `LiveMap`, to identify objects reached during traversal, improving both virtual memory performance and ability to reclaim storage with respect to Bartlett's algorithm, which instead preserves all objects in promoted pages, whether live or not. We have measured improvements between 20% to 40% in the number of calls to `scavenge()` and in overall memory usage.

The algorithm of the collector is as follows:

1. Clear the `LiveMap` bitmap
2. Scan the root set to determine objects that cannot be moved. Any object directly reachable from a root is marked as *live* setting a bit in the `LiveMap` bitmap and the page to which it belongs is promoted to *ToSpace*
3. Scan *ToSpace*, i.e. each promoted page sequentially, looking for objects marked *live*. Traverse each such object by applying the following procedure to each pointer it contains:
   (a) if it points to an object in another heap: traverse the object.
   (b) if it points to an object not marked *live* in a promoted page: mark the object *live* and, if it is earlier in the page being scanned or in an already scanned page [1], traverse it.
   (c) if it points to an object not marked *live* in a non promoted page: scavenge the object, i.e. copy it, mark it and the copy as *live*, set a forwarding pointer within the object to the copy and update the pointer to point to the copy.
   (d) if it points to an object marked *live* in a non promoted page: update the pointer to the value of the forwarding pointer stored within the object.

[1]To implement the distinction between scanned and unscanned pages, we use a fictitious space for scanned pages: when a page is scanned it is moved to this space and at the end of the collection all scanned pages are moved back into *ToSpace*. Our benchmarks have shown that by maintaining this distinction the number of calls and time spent in `scavenge()` are reduced in half.

If we discard cases (a) and (b), this is basically the Cheney copying traversal algorithm [14], which copies objects performing an iterative traversal of all nodes in a breadth-first order.

All new pages allocated for copying reachable objects belong to *ToSpace*, therefore a copied object is traversed when the collector examines its page.

Cases (a) and (b), needed to cope with multiple heaps and objects that need not be copied, introduce recursion. We did not consider worth while to complicate the algorithm to eliminate this recursion, because we expect case (a) not to occur often and case (b) not to produce a deep recursion [2].

In our design we have been able to reduce the space overhead required for each collectable object to just whatever C++ needs for implementing classes with virtual functions, eliminating the header used in Bartlett's implementation, which contained a forward bit, the size of the object and the identifier of a callback routine.

It is interesting to note that by using an appropriate page size our technique reduces to the *treadmill* algorithm by Baker [5]. This is due to the fact that objects larger than the page size are never moved, so it is sufficient to set the page size to a value smaller than any object (e.g. 8 or 16), to achieve the treadmill behavior: objects change space without moving. The doubly linked list for objects used in Baker's algorithm is realised in one direction by the queue of pages to be scanned that the CMM collector maintains, while the links in the other direction, needed to splice new elements into the treadmill when an object moves from one space to another, are replaced by the table of pages that is used by CMM to determine which pages are free.

5.1 Almost Generational Collection

Copying all live objects at each collection turns out to be too costly, according to our experiments. Therefore the collector for the `DefaultHeap` implements the generational technique of Bartlett [8], with a few variations in the numbering schema and in the handling of copied objects between a generational and a full collection.

There are logically three spaces: *FromSpace*, *FreeSpace*, and *OldSpace*. As before, new objects are allocated in *FromSpace*, collection promotes some pages from *FromSpace* to *OldSpace* and moves live objects from *FromSpace* to *OldSpace*, tracing but not touching objects already in *OldSpace*, then *FromSpace* is merged into *FreeSpace* and *FromSpace* is restarted as empty. As far as possible new pages needed for *FromSpace* are taken from *FreeSpace*. Once in a while, when generational collection cannot recover a certain percentage (65% by default) of available memory, a full collection is forced, by merging *OldSpace* into *FromSpace*. When this step is performed, the algorithm reduces to the non generational case.

To implement the three logical spaces, the space-identifier for pages is used. A counter `fromSpaceID` is maintained, which starts at 3 and is incremented after each collection. *FromSpace* is represented by pages with space-identifier equal to `fromSpaceID`, *OldSpace* is represented by pages with 0 space-identifier, *FreeSpace* consists of the remaining pages. During collection, objects are copied to pages, either new or recycled from *FreeSpace*, whose identifier is set equal to 0, thereby extending *OldSpace*.

Merging *FromSpace* with *FreeSpace* is achieved just by incrementing `fromSpaceID`, while merging *OldSpace* into *FromSpace* requires changing the space-identifier of pages in *OldSpace* to `fromSpaceID`.

The technique is "almost" generational since it does still scan the *OldSpace* in order to identify live objects and does not implement a *write barrier* [43] that would allow the creation of pointers from *OldSpace* to *FromSpace* to be detected and to limit to those pointers the start of search for live object in *FromSpace*. Nevertheless the technique turns out to be quite effective to reduce the amount of copying by the collector, as indicated by the data we report in the section on performance.

6 Multiple Heaps

Besides the copy-collected heap, also the traditional uncollected heap must be present for use with the primitives `malloc()` or `new` on uncollected classes. Programs and libraries that use uncollected objects in

[2]In our benchmarks the number of pages promoted in the promotion phase is typically between 10 and 20 on an overall heap of 5000 pages. The depth of recursion is the length of a chain of pointers between promoted non scanned pages. In the average case, when half of the promoted pages have been scanned and within each of them one half has been scavenged, the probability that there is a chain of length n is on average $(\frac{1}{4} \cdot \frac{20}{5000})^n$ which is 10^{-9} for a length of 3.

an unsafe way for the collector [22] must use the uncollected heap or a non-copying heap if they use objects that can't be relocated.

Besides the predefined heaps, the CMM allows users to build their own heaps with specific allocation strategies for their applications.

When dealing with multiple heaps, some essential requirements must be fulfilled:

- *allowing pointers across heaps*: restricting the range of pointers is difficult and inconvenient.
- *transitivity of liveness*: if an object is pointed to by a live object it is live as well. We must ensure that a pointer crossing heap boundaries does not go unnoticed by the collector.
- *independence of collectors*: it must be possible to write a collector for a particular heap, without relying on the collectors for other heaps, provided the root set for this heap is known.

In Figure 3 three heaps are present: the uncollected, the copy collected (default), and one user collected heap.

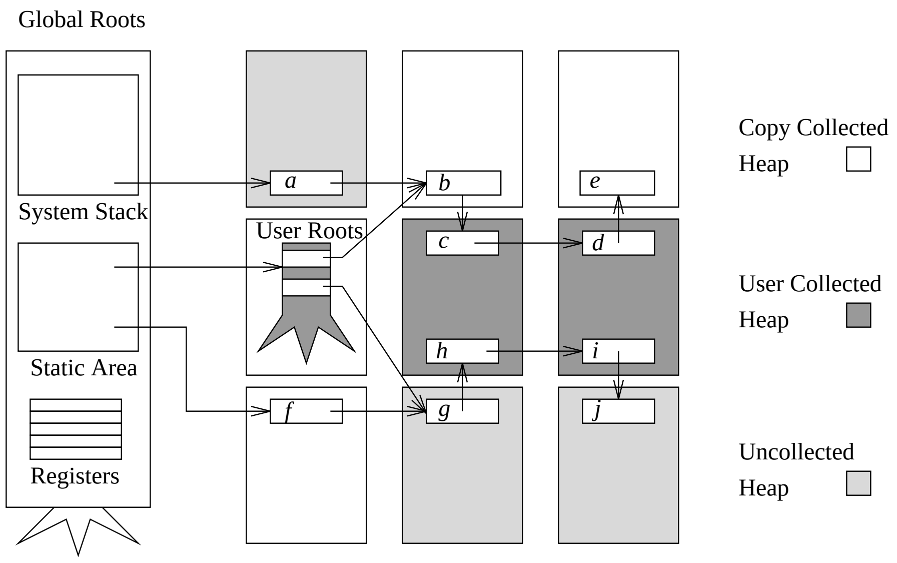

Figure 3: Multiple heaps

All six possible cross-heap pointers are shown. The user heap is maintained by the user, who keeps a record of the roots into his heap, so that he can perform a collection of that heap when appropriate, without involving a global collection. However the default collector must be capable for instance of identifying e as a live object, even though this involves passing through several heaps.

Handling pointers across heaps is a critical issue. In the present CMM framework, collections are performed separately for each heap. Collection on a heap is normally triggered when space is exhausted on that heap. There are several reasons for this:

1. the order of operations is relevant: the mostly-copying collector requires that all roots be scanned before it can start the promotion phase.
2. a single global traversal would require collectors to be aware of each other or to implement a schema to transfer control among each other.
3. certain heaps are designed to perform a partial collection limited to their areas: doing a full tracing of all memory would defeat the purpose of a generational collector or a collector for temporary objects.

4. each collector maintains bookkepping information: having all such structures at once may require too much memory.

The drawback of this approach is that collector costs may increase in applications that fill frequently more than one heap at a time.

Having separate collections for each heap means delegating to each heap the responsibility of dealing with cross-heap pointers. We describe here the protocol for CMM and how outgoing pointers are handled. Incoming pointers are an issue for heaps that require a special handling of ambiguous roots: we will discuss the solution for the mostly-copying heap.

6.1 Protocol

The behavior of a CMM collector can be described in terms of tricolor marking [19]. At the beginning of a collection all objects are *white*, while at the end the objects that will be retained must be *black*. The trace phase of the collector changes each white object it encounters into a *gray* object. A gray object is an object that must survive the collection, but may contain pointers to white objects. A gray object is turned to black when all pointers it contains no longer point to white objects, i.e. it may point only to either gray or black objects. The invariant that must be preserved throughout the collection is that no black object holds a pointer directly to a white pointer.

The protocol for CMM consists in the two methods `traverse()` and `scavenge()`. `traverse()` is used to trace live objects. Its code is fixed for each class of collectable objects and depends only on the structure of the class: `traverse()` is implemented as a virtual member function for each class of collected objects and section 9 describes how to synthesise its code. `traverse()` is applied to a gray object and turns it into a black object by applying `scavenge()` to all its pointer locations. `scavenge(loc)` is the method used to turn a white object `wobj` referred by `loc` into a gray one `gobj`.

So the postcondition for `scavenge(loc)` is that `loc` should contain `gobj`, which must be isomorphic to `wobj`, and information should be recorded so that any further call to `scavenge(loc1)`, where `loc1` holds a pointer to `wobj`, will replace it with a pointer to `gobj`.

For a mark-and-sweep collector scavenging may consist in just setting the mark bit for the object, while for a copying collector it entails moving the object to *ToSpace* and storing somewhere a forwarding pointer to it.

6.2 Outgoing pointers

To achieve coordination among collectors for the various heaps, one has to agree to a mechanism that allows traversing objects in different heaps on behalf of the collector for another heap. While traversing a foreign heap, a collector should not be allowed to make changes to the objects it visits, except to update recognised pointers to an object in its own heap, after the object has been moved.

This means that one must perform scavenging only for objects in the heap being collected. In other words the scavenge procedure must remain the same throughout a collection, but the scavenge for one heap must not operate on objects in other heaps. `scavenge()` is implemented a member function of each heap class, while `traverse()` is a member function of each class of objects.

Figure 4 illustrates the interplay between `scavenge()` and `traverse()`:

Suppose a garbage collection is started in heap A that uses a copy collector. While traversing object $A1$, the garbage collector identifies a pointer to the object $B1$, belonging to heap B. Object $B1$ is scavenged by the `scavenge()` function of the heap A. This function recognises object $B1$ as external to heap A, so it does not copy the object, as it would if it were internal to the heap, but only traverses the object to determine whether further objects in heap A can be reached from it. The behaviour of `scavenge()` changes again when object $A2$ is reached, which belongs to heap A. Applying the scavenge function of heap A has the effect of copying object $A2$.

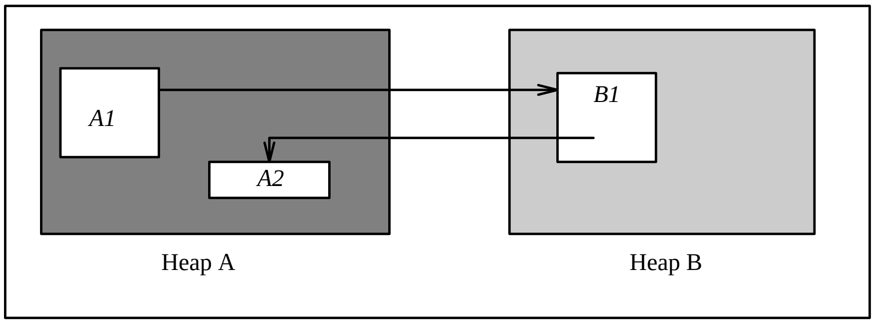

Figure 4: Pointers across heaps

6.3 Mostly-copying with multiple heaps

The mostly-copying collector algorithm described previously relies on the fact that all locations that might contain ambiguous pointers are known in advance: they coincide with the root set. Therefore pages to be promoted can be identified by a single linear scan of the root set, in the promotion step of the algorithm.

However, when multiple heaps are present and pointers across heaps are allowed, ambigous pointers might be detected at a later stage.

Requiring that such pointers be registered as roots is not practical, since it would entail registering as root any collected object that is passed to an external library procedure, which might store such pointer internally. This can be cumbersome to do and may be accidentally forgotten.

Modifying the promotion step of the algorithm to perform a complete traversal from the root set in order to identify ambiguous pointers to collected objects, would be a costly solution with live objects being traversed twice.

In [2] we proposed a slight variation to our basic algorithm. Instead of updating pointers immediately when an object is copied, just the location to be updated is recorded, using a temporary bitmap. If it is later discovered that the object should have not been moved, but rather the page should have been promoted, all the objects in such page are restored from their copies. The updates to pointers are performed only at the end of the algorithm, using the bitmap and the forwarding pointer stored in the objects. This technique is similar to the one suggested by Detlefs [16] to handle C/C++ unions of pointers and non-pointers.

The current implementation uses a simpler solution which distinguishes between *opaque* and *transparent* heaps.

A transparent heap contains only traced `CmmObject`'s and stores its roots in global variables, while an opaque heap may contain other kind of objects that are not traceable. For instance the `UncollectedHeap` will be opaque, while the `DefaultHeap` will be transparent.

Since objects in opaque heaps cannot be accurately traced, when collecting one heap H, all other opaque heaps are supposed to contain ambiguous pointers into H. In the page-promotion phase of the copying collector, each page belonging to an opaque heap is scanned conservatively to identify locations containing pointers into the copying collector heap. All such locations are dealt as ambiguous roots, and the page they point into is promoted. Vice versa, in the traversal phase, if an object contains a pointer into an opaque heap, such pointer is not traversed.

The cost of collection for the `DefaultHeap` is proportional to the size of opaque heaps plus the amount of live objects in the `DefaultHeap`. It is a fairly common case that no pointer to traced objects is stored in objects dynamically allocated in opaque heaps or these objects are registered explicitly as roots: for this case CMM provides an initialisation option to disable scanning opaque heaps.

This solution for pointers across heaps has however the drawback that it cannot recover garbage cyclic structures that span across transparent and opaque heaps. More sophisticated techniques like those used in distributed garbage collectors [33], would be required to handle this case.

Even if one could envision collections being done concurrently on all heaps, we have not investigated this aspect.

Vice versa, a multithread version of CMM has been developed by C. Heckler at the University of Paderborn, using Solaris threads: a single heap is shared among the threads.

7 The CMM Framework

Heap memory consists of pages of equal size. The allocator for each heap requests pages for storing objects from the low level page allocator. Each page is tagged with the heap to which it belongs.

CMM uses a bitmap to deal with internal pointers to objects. Whenever a CMM object is created, the bit corresponding to its first word is set. Using this information, a pointer inside that object can be normalised to the beginning of the object, simply scanning the bitmap backward until the first set bit is found.

When an object is moved, its first word is replaced by a forwarding pointer to the new object. There is no need to use a special mark to recognise forwarded objects, as in other implementations, since the collector can determine this situation from the fact that the object is marked *live* and its page is in *FromSpace*.

7.1 Class `CmmObject`

The run time support required for collectable objects is provided by the class `CmmObject`. Every class of collectable objects must be derived from `CmmObject`, which must appear first in a base list for multiple inheritance. `CmmObject` provides an overloaded `new` operator that takes care of allocating the object in a specified heap. Function `traverse()` is used to trace accurately objects allocated in the `DefaultHeap`. The other member functions of `CmmObject` are useful in the implementation of collectors and less relevant for normal users of the CMM:

```
class CmmObject
{
public:
  void* operator new(size_t, CmmHeap* = Cmm::heap);
  virtual void traverse();
protected:
  CmmObject* next();              //  returns the next adjacent object
  bool forwarded();               //  tells whether the object has been forwarded
  void setForward(CmmObject* ptr); //  sets the forwarding pointer
  CmmObject* getForward();        //  returns the forward location of the object
  Heap* heap();                   //  returns the heap to which the object belongs
  void mark();                    //  mark object as live
  bool marked();                  //  tells whether the object is marked
};
```

7.2 Object Creation

When creating a collected object one can specify in which heap to allocate it. The parameter `heap` can be supplied in the standard C++ placement syntax for the `new` operator:

```
  p = new(heap) Person(name, age);
```

If the user does not specify any heap, the default heap `heap` is used:

```
  p = new Person(name, age);
```

which is equivalent to:

```
  p = new(Cmm::heap) Person(name, age);
```

where `Cmm::heap` is a static member variable initialised to `Cmm::theDefaultHeap`.

When creating collected objects, the programmer can decide case by case where to allocate them. In summary, the following are the alternatives for object allocation:

Heap	**Classes**	**Creation**
`UncollectedHeap`	uncollected	`::new` or `malloc`
`DefaultHeap`	collected	`new` or `new(Cmm::theDefaultHeap)`
`MarkAndSweep`	collected	`new(Cmm::theMSHeap)`
user defined	collected	`new(heap)`

where we call collected those classes that inherit from `CmmObject` and uncollected all others.

With the CMM, object allocation is not tied to the type of an object as in other proposals, so a programmer can design his classes without committing to a particular memory policy. The policy can be decided later, or even be different in different portions of an application. For instance, in the PoSSo polynomial systems solver, one sets the variable `heap` to the heap implementing the stack policy before starting a simplification step. Throughout the simplification, all objects (monomial, polynomial, large precision integers, lists and so on) are allocated in this heap and freed in a single step at the end of the simplification. After simplification, one reverts to the normal heap.

It is essential that this can be done without changing a single line in the user code. This is not a just a program development issue: sensibly different versions of the same program would have to coexist in one application if different policies required different code.

As usual, some coding convention must be adopted in the use of the global variable `Cmm::heap`: either each routine that sets it takes care of restoring it or a stack of heap pointers can be defined, so that after finishing using one heap one can revert to the previous heap.

7.3 Array of `CmmObject`'s

For creating arrays of `CmmObject`'s CMM exploits overloading of the `new[]` operator for allocating arrays of objects, a mechanism recently introduced in the ANSI C++ standard.

Class `CmmArray` can be used to create arrays of `CmmObject`'s as follows. To create an array of objects of class `Item`, the programmer must supply the following code to overload the `new()` operator for class `Item`:

```
void*
Item::operator new[](size_t size)
{
  return sizeof(size_t) + (char*)new(size) CmmArray<Item>;
}
```

Then arrays of `Item` can be created normally, for instance:

```
Item* anArrayOfItems = new Item[20];
```

The constructor for class `Item` with no argument will be called for each `Item` in the array. The operator `[]` can be used to access such arrays, e.g:

```
anArrayOfItems[i].print();
Item anItem = anArrayOfItems[3];
```

8 Heap Classes

To manage a heap one normally has to maintain the set of roots for the objects in the heap, manage the pages where objects are allocated, and implement the memory allocation and recovery primitives. A suitable encapsulation for these functionalities is provided by the `Heap` class.

8.1 Class `CmmHeap`

A class implementing a heap must supply definitions for the following pure virtual functions: `alloc()` and `reclaim()`, implementing the memory allocation strategy, `collect()` to perform collection, and `scavenge()`, the action required to preserve live objects encountered during traversal. Heap classes are derived from the abstract class `Heap`, defined as follows:

```
class CmmHeap
{
 public:
  CmmHeap();                              //  initialiser
  virtual CmmObject* alloc(int bytes) = 0;
  virtual void reclaim(void*) = 0;
  virtual void collect() = 0;
  virtual void scavenge(CmmObject **ptr) = 0;
  bool inside(CmmObject* ptr);            //  checks whether ptr is within this heap
 protected:
  int reservedPages;
  RootSet* roots;
};
```

The variable `reservedPages` represents the number of pages allocated to the heap. `roots` points to an instance of class `RootSet`, used for registering roots into this heap.

8.2 Predefined Heaps

The CMM provides four predefined heap classes:

- `DefaultHeap`: implements a mostly-copying generational collection discipline;
- `UncollectedHeap`: provides the traditional manual allocation discipline. It is available through the default `new` operator or the functions of the `malloc` library. Objects not inheriting from `CmmObject` are allocated in this heap. There are two ways to use it:
  - by using the standard global `::new` operator;
  - by using `new(Cmm::theUncollectedHeap)` for a class derived from `CmmObject`.
- `TempHeap`: this heap is useful when within a phase of a program large amounts of temporary objects are allocated which can be discarded at the end of the phase.
- `MarkAndSweep`: implements a mark-and-sweep policy. It is a fully conservative collector that does not require any information on the layout of objects.

9 Using the `DefaultHeap`

In order to store objects in the `DefaultHeap`, programmers must derive their classes from class `CmmObject` and define the method `traverse()` for such classes. In case of multiple inheritance class `CmmObject` must always be the first in the base list. `traverse()` must be defined according to precise rules which ensure that collected objects or pointers to them contained in the class or in its superclasses are correctly identified by the collector. These rules are a generalisation of those in [8].

Denoting with *CmmClass* a class derived from `CmmObject`, the code of function `C::traverse()` for a class `C` must contain:

(a) `scavenge(&x)`, if `C` contains a pointer to a collected object, say `class C {` *CmmClass* `* x; }`

(b) `x.traverse()`, if `C` contains an instance of a collected object, say `class C {` *CmmClass* `x; }`

(c) *CmmClass*`::traverse()`, if `C` is a class derived from another collected class, say `class C:`*`CmmClass`* `{...}`

(d) `scavenge(CmmRefLoc(C, x))`, if `C` contains a reference to a collected object, say `class C {` *`CmmClass`* `&x; }`

(e) `scavenge(VirtualBase(`*`CmmClass`*`))`, if `C` derives from a virtual base class, say `class C: virtual` *`CmmClass`* `{...}`.

The last rule is necessary to deal with virtual base classes, identifying the hidden pointer to the base class, present inside an object. `scavenge()` takes care of updating this pointer to the new location of the object after it has been copied. The hidden pointer cannot be identified in a compiler independent way, so the CMM provides a macro `VirtualBase`, which is compiler specific. For instance, its definition for the GNU C++ compiler is:

```
  #define VirtualBase(class)  &(_vb$ # class)
```

and for the CFRONT compiler:

```
  #define VirtualBase(class) &(P ## class)
```

The following example illustrates the rules. Suppose the following collected classes are defined:

```
class Link: public virtual CmmObject      //  Rule (e) applies here
{
  CmmObject *next;                        //  Rule (a) applies here
  void traverse();
};

class Monomial: public virtual CmmObject  //  Rule (e) applies here
{
  long coefficient;
  Term term;                              //  Rule (b) applies here
  void traverse();
};

class Polynomial: public Monomial,        //  Rule (c) applies here
                  public Link             //  Rule (c) applies here
{
  void traverse();
};
```

Class `Link` inherits from `CmmObject`, therefore its `traverse()` must be invoked. Since inheritance is virtual, a `Link` object contains an hidden pointer to the virtual base class `CmmObject`, which must be scavenged. Finally, in the field `next`, a `Link` stores a pointer to a collected object that needs to be scavenged. Therefore `traverse()` becomes:

```
void Link::traverse()
{
  CmmObject::traverse();                  //  Appling rule (c)
  scavenge(VirtualBase(CmmObject));       //  Applying rule (e)
  scavenge(&next);                        //  Applying rule (a)
}
```

Also `Monomial` uses virtual inheritance from `CmmObject`, which is dealt similarly in `traverse()`. Since a `Monomial` contains a `Coefficient` in `term`, this object must be traversed as well:

```
void Monomial::traverse()
{
  CmmObject::traverse();                //  Appling rule (c)
  scavenge(VirtualBase(CmmObject));     //  Applying rule (e)
  term.traverse();                      //  Applying rule (b)
}
```

Since `Polynomial` inherits from classes `Monomial` and `Link`, its code for `traverse()` should be:

```
void Polynomial::traverse()
{
  Monomial::traverse();                 //  Appling rule (c)
  Link::traverse();                     //  Appling rule (c)
}
```

Preprocessing [20] or compiler support [36] could be adopted to avoid hand coding of these functions and risks of subtle errors in programs. Notice that the preprocessor could be a very simple code analyzer which takes as input the header files, macroexpanded by the standard preprocessor, and produces the code for the traverse methods for each class derived from `CmmObject`. Such code needs just to be added to the source code for normal compilation and this step is only required when header files are changed, therefore with minimal interference with the normal tools for software development. Another possibility would be to obtain layout information for objects from compiler debug output, as in Wilson's type descriptor generator system, even though this technique is less portable [28].

P. Beard [9] suggested the use of C++ pointers to member for implementing `traverse()`. Each class would contain a static table with pointers to members of the class that are pointers to collected objects. There is a single `traverse()`, which uses this table to invoke `scavenge())` on each member. However the implementation of pointers to members is not not very efficient: for instance on a SPARC it requires 11 machine instructions instead of one for each invocation of `scavenge()`.

10 `MarkAndSweep` Heap

While a copying collector needs information about the internal layout of objects, a mark-and-sweep collector can be fully conservative in tracing objects and therefore can do without such information. In this case a mark-and-sweep collector can be simpler to use, but on the other hand, it may not be able to identify all garbage. The amount of unreclaimed garbage may vary depending on the specific application and the data structures used and is quite hard to estimate.

For instance, if the application uses a large number of linked objects, a value in a traced object mistakenly considered as a pointer may lead to retain an object and consequently all objects connected to it, for a quite significant overall amount of space. In the section on performance we report about cases when this phenomenon had quite visible effects.

A mark-and-sweep collector may also be subject to memory fragmentation, especially if the program uses objects of many different sizes [44]. Recent measurements by Zorn [18] and Wilson [45] suggest however that a good non compacting allocator can limit fragmentation to acceptable proportions.

10.1 Using the `MarkAndSweep` Heap

Also objects allocated in the `MarkAndSweep` heap must belong to a subclass of `CmmObject`.

As usual, you can select the `MarkAndSweep` as the default heap by setting the static variable `Cmm::heap`:

```
  Cmm::heap = Cmm::theMSHeap;

  // BigNum is a subclass of CmmObject, so b will be allocated in the MarkAndSweep heap.
  BigNum* b = new BigNum("1231345345345345");
```

Objects in the `MarkAndSweep` heap can be recycled immediately by releasing their memory with `MarkAndSweep::reclaim()`. Programmers are not encouraged to use this feature, since its use is often error prone and somewhat contrary to the spirit of a garbage collector. Nevertheless using `reclaim()` appropriately may delay the invocation of the garbage collector, especially when large data structures are involved.

10.2 Implementation Issues

The `MarkAndSweep` collector handles small and large objects differently. A small object is an object whose size is less than 256 bytes. Since many applications use mainly small objects (for instance list nodes), the allocator tries to optimise allocation and deallocation for such objects.

The `MarkAndSweep` allocator requests pages to the low level CMM page allocator. Each page is marked as either *single*, when used for small objects, or *grouped*. If an object is bigger than the page size, several contiguous *grouped* pages are joined together to store the object.

Each *single* page contains only objects of the same exact word size. This eliminates the need for storing size information for each object (since the size information is stored in the page header) and simplifies scanning such pages, for example during the sweep phase. A page can be recycled for use with objects of a different size only after the `MarkAndSweep` collector has freed it completely.

For each page, a free list linking all free objects within that page is maintained. Such list is updated when an object is allocated or released.

The collection algorithm consists of a mark and a sweep phase. During the mark phase live objects, traced from the root set, are marked as well as the pages that contain them. During the sweep phase any non marked object is released: in the case of a small object, it is added to the free list of the page; otherwise the corresponding set of *grouped* pages is added to the set of available pages. Non marked pages can be immediately released without scanning them. Such an optimisation can be quite effective with applications that produce large amounts of small garbage objects.

11 An Example of Customisation

We present here an example of the kind of customisation of the collector that is feasible with the CMM. The solution is fairly simple to realise with the support provided by the CMM and provides significant performance improvements as reported in a later section.

11.1 The Case Problem

The Buchberger algorithm is an algorithm in computer algebra that computes the Gröbner basis of an ideal of polynomials, with a technique that is reminiscent of the Knuth-Bendix algorithm for term rewriting systems. Quite briefly, the algorithm is given a set of multivariate polynomials, and it must generate its Gröbner basis, i.e. the set of the smallest polynomials according to a certain ordering, such that any of the original polynomials can be obtained as a linear combination of its elements.

Each step of the algorithm tests whether one candidate polynomial `p` is a combination of the elements of the basis computed so far, the `simplifiers`. This step is called a reduction since it repeatedly tries to simplify the polynomial to determine whether it reduces to zero. If the resulting polynomial is not zero, it is inserted in the basis. Candidate polynomials are produced by `Spolynomial` for each pair of simplifiers. This and the fact that reduction with one simplifier requires repeating simplifications with the other polynomials, account for the exponential space growth in the computation, which stresses heavily many systems and in particular the memory management. Several real problems involve weeks of computation and memory in the order of gigabytes.

Here is a fragment from the actual code performing the reduction step for a pair of simplifiers:

```
Poly* p = pair->SPolynomial();

if (p != 0) {
  p = simplify(p, simplifiers);
```

```
    p = normalize(p);
  }
```

The relevant aspects of the algorithm with respect to memory management are:

1. Large amounts of memory are allocated during `simplify()` and most of this memory can be freed at the end of this step. The only data to be preserved is the simplified polynomial that must be inserted into the final basis.
2. In many cases `simplify()` returns a zero polynomial. In these cases no memory must be preserved.
3. Since the complexity of the algorithm is exponential, the amount of memory allocated by `simplify()` also grows exponentially with the size of the ideal.

11.2 Solution with the CMM

Having identified portions of the computation where objects with only limited lifetime are created, we can segregate these objects from the rest and reclaim their space soon after their lifetime expires, without requiring a full collection.

In order to exploit the characteristics of the Buchberger algorithm, we use two separate heaps: the default one and a special one for this algorithm (`simpHeap`), an instance of the class `TempHeap`.

Memory is usually allocated inside the default heap, but before calling `simplify()` the heap is switched to the `simpHeap`. All the memory allocated during `simplify()` is therefore obtained from the `simpHeap`.

Notice that this does not require any changes to any of the remaining functions in the application: the algebraic operations on polynomials, coefficients etc. are unmodified and use the standard `new` operator to allocate objects.

After returning from `simplify()` we switch back to the default heap, and the polynomial returned by `simplify()` is copied into the default heap. At this point the `simpHeap` contains no live data and can be freed with a single operation without involving a garbage collection.

Here is the code again augmented with instructions for CMM memory management.

```
  CmmHeap* previousHeap = Cmm::heap;  //  save the current heap
  Cmm::heap = simpHeap;               //  set the current heap to simpHeap
  Poly* p = pair->SPolynomial();
  if (p != 0) {
    p = simplify(p, simplifiers);
    p = normalize(p);
  }
  Cmm::heap = previousHeap;           //  restore the previous heap
  p = new Poly(*p);                   //  copy p out of the simpHeap
  simpHeap->clear();                  //  empty the simpHeap
```

Notice that the copying constructor for `Poly` has been specialised to perform a deep copy of the whole structure.

Emptying the `simpHeap` involves just resetting a few internal variables.

This simple solution has the drawback that the `simpHeap` might grow too much during `simplify()`. This may happen if one simplification step is very complex and lots of garbage is generated.

So we must provide the class `TempHeap` with its own collector, for which we choose a copying strategy. With the facilities provided by the CMM, implementing this collector is a fairly simple task. The `TempHeap` manages two semispaces of variable size, consisting of series of contiguous blocks of memory. The collector copies the roots into `toSpace`, and then scans the `toSpace` further copying into `toSpace` objects reachable from there.

Here is the code for the collector, which relies on support provided by class `CmmObject` for skipping from one object to the next consecutive one in memory (`next`) and by the user in the form of a `traverse()` method for each class of collectable objects.

```
TempHeap::collect()
{
  CmmObject* objPtr;
  //  check that it is worth collecting
  if (fromSpace->current < size * 0.8)
    return;
  //  new objects are allocated in fromSpace
  //  swap semispaces before copying
  swap(fromSpace, toSpace);
  roots.scan();                 //  copy all roots
  objPtr = fromSpace->bottom();
  while (objPtr < fromSpace->current()) {
     objPtr->traverse();
     objPtr = objPtr->next();   //  advance to next object
  }
  toSpace->reset();
  expand();                     //  expand heap if needed
}
```

The collector for the `TempHeap` is most effective if its roots can be identified without a search. In our case we:

- avoid conservative tracing of the stack by calling the collector "opportunistically" [42] within `simplify()`; [3]
- avoid tracing other areas of memory since no pointers to objects in the `simpHeap` are handed out to procedures that store them elsewhere;
- know which objects in the `simpHeap` are in use within `simplify()`, which are consequently the only roots for the `simpHeap`.

Before starting the simplification we register as roots for the `TempHeap` the two variables that refer to the only objects in `simpHeap` used within `simplify()`: the variable containing the current polynomial and the one containing the current monomial.

The place for the opportunistic call to the collector is after each reduction step. An actual collection takes place only if the space left in the heap is below a certain percentage threshold. The garbage collector visits the two registered roots and copies all objects reachable from them. In practice the current polynomial and the current monomial are copied into `toSpace`. At the start of the next reduction cycle a whole semispace is emptied and available for further allocation.

Here is a sketch of the code, where `SL->first` accesses the first element of the list of polynomials `SL`, `SL->next` accesses the rest of such list and `simplifier->head` selects the head monomial of polynomial `simplifier`:

```
Poly* simplify(Poly* p, PolyList &simplifiers)
{
  if (simplifiers == NULL)
    return p;
  CurrentPolynomial = p;
  CurrentMonomial = *p;
  //  register the roots into this heap
  simpHeap->roots.set(&CurrentPolynomial);
  simpHeap->roots.set(&CurrentMonomial);
```

[3]The technique can be exploited also in a multithreading application, if one can arrange to use a separate `TempHeap` for each thread.

```
  while (CurrentMonomial != 0) {
      bool reduced = false;
      //  iterate through the list of simplifiers
      PolyList SL;
      for (SL = simplifiers; SL != NULL; SL = SL->next) {
          Poly* simplifier = &SL->first;

          if (divisible(CurrentMonomial, simplifier->head)) {
              CurrentMonomial = reduce(simplifier);
              reduced = true;
              simpHeap->collect();    //  invoke the collector
              break;                  //  restart reductions
            }
        }
      if (!reduced)
         CurrentMonomial = &CurrentMonomial->next;
    }
  //  unregister the roots
  simpHeap->roots.unset(&CurrentPolynomial);
  simpHeap->roots.unset(&CurrentMonomial);
  return CurrentPolynomial;
}
```

12 Extending the Framework

This section illustrates how the CMM framework can be extended by implementing new heaps as classes derived from `CmmHeap`. We describe the mechanism by completing the example of the previous section. It is a simplified version the actual `TempHeap` supplied with the CMM, where two semispaces are used, which are just fixed size blocks of contiguous memory.

First we define the `TempHeap` class as a `CmmHeap` consisting of two areas that implement the `fromSpace` and the `toSpace` of the collector:

```
class TempHeap: public CmmHeap
{
  public:
   TempHeap(int);

  private:
   unsigned fromSpace, toSpace;
   unsigned fromTop, toTop;
   int size;
   CmmObject* copy(CmmObject*);
};
```

The creation of a `TempHeap` involves requesting two groups of pages for the two spaces:

```
TempHeap::TempHeap(int bytes)
{
   size = bytes;
   fromSpace = allocatePages(bytes / BYTESperPAGE, this); fromTop = 0;
   toSpace = allocatePages(bytes / BYTESperPAGE, this);
}
```

Allocating memory for an object consists just in advancing the index `fromTop`:

```
CmmObject* TempHeap::alloc(int bytes)
{
   if (bytes > size - fromTop) error("No more memory");
   CmmObject *obj = (CmmObject*)(fromSpace + fromTop);
   fromTop += bytes;
   return obj;
}
```

The collector uses the root set to traverse the roots. After having moved to `toSpace` all the objects reachable from the roots, it traverses those objects in order to move all further reachable objects. As the final step the collector exchanges the roles of `fromSpace` and `toSpace`.

We have shown earlier the code for `TempHeap::collect()`, which invokes the method `traverse()` to trace collectable objects. `traverse()` performs scavenging of objects by means of method `scavenge()`. Each class of heap has its own strategy for scavenging objects: for a copying collector the strategy is to copy the object, while for a mark-and-sweep collector it will consists in marking the object.

The specific action required for scavenging objects in the `TempHeap` is as follows:

```
void TempHeap::scavenge(CmmObject** ptr)
{
  CmmObject* p = basePointer((GCP)*ptr);  //  find the start of the object
  int offset = (char*)*ptr - (char*)p;
  if (!inside(p))
    visit(p);                             //  traverse objects in other heaps
  else if (p->forwarded())
    //  update pointers to forwarded objects
    *ptr = (CmmObject*)((char*)p->getForward() + offset);
  else {
    CmmObject* newObj = copy(p);
    p->setForward(newObj);               //  store forwarding pointer
    *ptr = (CmmObject*)((char*)newObj + offset);
  }
}
```

Unless the object had been copied already, it is copied in the next semispace and a forwarding pointer left with the original.

13 Performance

There are several questions on performance that we tried to answer through benchmarking: whether the CMM, which uses C++ to allow customisation, loses in performance with respect to a collector written purely in C; how a copying collector compares with a mark-and-sweep collector; what benefits a specialised collector provides; what benefits "opportunistic" scheduling of collections [42] may offer.

The Buchberger algorithm appears quite suitable for benchmarking since it provides a wide variety of behaviours arising from different input data: from cases where quite large objects are generated, to cases where a large number of small objects are generated.

The benchmarks were run on a SparcStation 10 with 32 Mbytes of physical memory. The compiler used was `gcc` version 2.7.2. Profiling information was analyzed by means of `gprof` from the call graph profile produced during execution. All times reported were computed to an accuracy of microseconds and are expressed in seconds, memory in Kbytes. The values reported are averages of 10 runs, with marginal differences between runs, typically within 2%.

The benchmarks were repeated several times over the years, since while we were developing the CMM, the implementation of the Buchberger algorithm was also being improved and even the C++ compilers evolved. Such improvements appeared to effect uniformly performance of the algorithms within the various collectors.

The suite of benchmarks compares the performance of the original Bartlett's implementation, the CMM using the default heap, the CMM using the `TempHeap` (denoted as CMM-TH in these tables), the CMM using the the `MarkAndSweep` heap (denoted as CMM-MSW) and two configurations of the Böhm-Weiser collector version 4.11, the standard configuration (denoted as Böhm) and a configuration with interior pointer recognition disabled, as discussed later (denoted as Böhm-NIP).

We used various sets of input polynomials, which are mentioned with the names they are given in the literature on symbolic algebra.

The results are summarised in Table 1.

Table 1: Benchmark Execution Times (sec.)

Bench	**Bartlett**	**CMM**	**CMM-TH**	**CMM-MSW**	**Böhm**	**Böhm-NIP**
katsura5	3.24	3.13	2.43	3.03	3.43	3.14
cohn1	12.45	7.36	5.54	7.14	7.35	7.01
valla	22.01	19.45	17.36	21.10	21.50	21.40
cyclic6	34.95	23.78	16.50	20.54	24.59	26.11
hawes4	217.77	163.86	131.30	168.19	166.79	156.09
katsura6	338.03	263.14	192.19	215.81	226.43	240.64
bjork70	1131.82	700.40	512.48	1048.53	967.23	623.47
katsura7	7415.85	5749.08	4941.91	>14000	Out of memory	5914.84

The improvement of using the `TempHeap` with respect to the default CMM heap appears to be significant across a variety of benchmarks, ranging from 15% to 36%. It is also interesting to notice that the CMM default algorithm has better performance than Bartlett's original, despite the overhead due to its use of C++ and member functions rather than straight C. This is mainly due to various code optimisations and code tuning.

With the mark and sweep collectors (CMM-TH and Böhm), the `katsura7` benchmark did not run to completion: the CMM-TH was stopped after 3.5 hours while Böhm collector run for almost an hour within 20 Mbytes of memory and then it started expanding very quickly until it exhausted 240 Mbytes of memory.

According to Böhm [12], mark and sweep conservative collectors seem to have a threshold of false pointer density, above which they don't work reliably. Once you get above this threshold, if you accidentally leak a Kbyte of main memory, it itself contains enough false pointers to cause at least another Kbyte leak, etc.

A large number of false pointers may arise when interior pointers are dealt as true pointers to objects. The Böhm collector can be configured not to consider, during the mark phase, pointers to the interior of an object as possible references to the object. In some cases this configuration choice can have significant impact on the amount of excess memory retention by the collector. In fact, by turning off interior pointer recognition in Böhm collector, the results in our benchmarks changed as reported in the column labeled Böhm-NIP.

The version of Böhm collector which does not check for interior pointers is generally unsafe and it use is not recommended, therefore it is discussed here only for comparison purposes. For many applications, and in particular some of the PoSSo applications [35] recognizing interior pointers is required, and in fact CMM always considers them as references to the enclosing objects.

To study in detail how much the garbage collector influences the overall performance, we analysed the various versions by means of a program profiler.

In Table 2 we report the results of running the benchmark `katsura6` [29], providing details on the timings of memory operations: `alloc`, the primitive allocator; `collect`, overall time spent in garbage collection; `pure alloc`, allocation time less collection time; `n. collect`, the number of calls to the collector; `avg. collect`, average time of a collection. In the last column we show two figures for each operation, one for the default heap and the second for the `TempHeap`, since both heaps are used. The profile of the `katsura6` benchmark is complex enough to be significant but also fairly typical of our applications.

Table 2: Execution Profile Analysis

	Bartlett	**CMM**	**CMM-TH**	**Böhm**	**Böhm incr/opport**
Katsura6 (profiled)	452.38	275.86	213.49	310	259.14
`alloc`	223.68	43.96	3.19+0.04	38.66	12.97
`collect`	215.07	37.06	2.47+0.03	23.94	43.10+4.80
`alloc - collect`	8.61	6.90	0.72+0.01	14.72	8.17
`number of collect`	931	450	16584+2	405	16625+17
`average collect`	0.23	0.08	0.00+0.01	0.04	0.00+0.28

For the CMM-TH case, we report separately the time spent in the TempHeap collector plus the time spent in the DefaultHeap collector. For comparison purposes we report in the last column also the execution analysis for Böhm collector used opportunistically as described in the next section: in this case the first number is due to the opportunistic calls and the second number to the normal calls to the collector. 43.10 seconds are spent in opportunistic collections, which occur independently of alloc and hence are not attributed to alloc, even though they represent costs related to allocation.

With the use of `TempHeap` the garbage collection time becomes negligible and accordingly allocation time is also significantly reduced. The total allocation cost using the default CMM heap is 44 seconds, which is slightly less than the gain from using the `TempHeap`. Therefore the 18% improvement in the overall execution time achieved by means of the `TempHeap` is close to the ideal maximum increase in performance one can expect to obtain by improving memory management.

We have also received satisfactory reports on the performance of CMM by the partners in the PoSSo project who used it in particular for implementing a linear algebra package [35].

For measuring the effectiveness of the generational schema used by CMM, we have produced an instrumented version of CMM. The instrumentation consists in adding a field in class `CmmObject` to record the identity of an object, and counting the number of times that an object gets moved.

Running our benchmarks with the instrumented version gave us an indication of the effectiveness of the generational schema of CMM: between 0.01% and 0.03% of all objects created in our application do actually get moved during collection, and the average number of copies per object ranges from 1.1 to 2.7.

13.1 Space Improvement

One of the design requirements for CMM was to be a compacting collector in order to reduce the requirements on virtual memory and swapping. This requirement was put forward by the mathematicians who use the Buchberger algorithm and who need to tackle problems that require several days of execution and quantities of memory in the order of gigabytes.

While it is expected that a copying collector might exhibit better space performance than a pure mark-and-sweep collector, such comparisons are quite hard to do through actual measurements. Moreover the CMM is only a partial copying collector, therefore the potential benefits of compaction might have been reduced.

Therefore we set up some benchmarks to measure the time and space performance of our collector in comparison of the same algorithm and the same code using the Böhm-Weiser collector [10], which is totally conservative and uses the technique of mark-and-sweep.

The Bartlett and CMM collectors use a similar policy for expanding the heap: the heap is expanded by the same fixed amount (1 MB), when after a collection the heap is more than 25% full, though this parameter can be changed through an environment variable. Böhm expands the heap by 1/4 of its size up to a maximum of 2 MB.

Table 3 presents the results of our measurements. The figures for memory in the tables represent the overall allocated memory, which however does not differ significantly from the value of the resident set size at the end of the program.

Table 3: Benchmark Memory Usage (Kbytes)

Bench	**Bartlett**	**CMM**	**CMM-TH**	**CMM-MSW**	**Böhm**	**Böhm-NIP**
hawes4	5466	5819	5820	7391	4752	3084
katsura6	4158	3051	4372	7531	3188	2584
bjork70	16338	12979	12964	9099	17344	8400
katsura7	26406	17519	10956	> 18900	> 240000	14656

In the bigger benchmarks, CMM indeed uses less memory than Böhm collector, and even more so when considering that approximately 1/3 of the amount reported is reserved to perform the copy and is otherwise unused. The CMM-TH is definitely faster and memory occupation is somewhat improved in these figures that show only the maximum amount of memory required in each execution.

In fact the improvement is more significant if we analyse how memory usage varies throughout the execution.

Such analysis explains for instance the apparent anomaly with `katsura6`, which requires 3 MB with CMM and 4.3 MB with CMM-TH. CMM-TH indeed uses less than 2MB throughtout the execution, except for a short period toward the end, when, during the collection of the `TempHeap`, the resulting polynomials are copied into the `DefaultHeap`. Since at that time most of the memory is used by the `TempHeap`, an additional 1MB is allocated for expanding the `DefaultHeap` and performing the copy. After this collection the memory used returns firmly below 2MB.

The following diagrams provide more details on the amount of memory used and reclaimed by CMM. Each bar represents every fifth collection. In the graphs "Used" represents the amount of data that survived a collection and "Reclaimed" shows the difference between the amount of data in use before and after a collection. Notice that one third of the allocated heap is used during collection to perform copy and is shown as **H**eadroom for copy in the diagram.

The following diagrams shows the execution using CMM-TH. In some cases the `TempHeap` is just cleared, while more often it needs to be collected: its emispaces are switched and very little data survives in it. This proves tat the `TempHeap` is mostly used for short lived temporary data. Even if these are good conditions for the applicability of a generational collector, the `TempHeap` has better performance than a generational collector, as reported in the next section. The `TempHeap` reduces significantly the overall amount of memory since space in the `TempHeap` is cleared and reused more often: this is feasible since the cost of collecting the `TempHeap` is small and so it does not have to be amortised over longer computation periods.

The diagrams show the situation after every tenth call to the `TempHeap` collector.

The following diagram shows the behaviour of the Böhm collector:

The difference in the amount of memory used with this collector appears due to excess retention because of ambiguous pointers. The data structures in the Buchberger algorithm are polynomials that may become quite large during the computation, and so significant amounts of memory may not be reclaimed if such structures can be reached through ambiguous pointers. The likelyhood of this phenomenon grows with the size of the problem and may significantly reduce the effectiveness of a conservative collector. To limit this problem Böhm collector has the possibility of declaring objects that do not contain pointers. The problem with ambiguous pointers is limited in the CMM since CMM knows the layout of objects and so it can be accurate in their traversal.

In some benchmarks the effect of such retention is more drastic:

The following diagrams shows the behavior of the Böhm collector when using `GC_malloc_atomic()` for allocating the coefficients of polynomials, which are multiple precision integers, implemented as large arrays of integers:

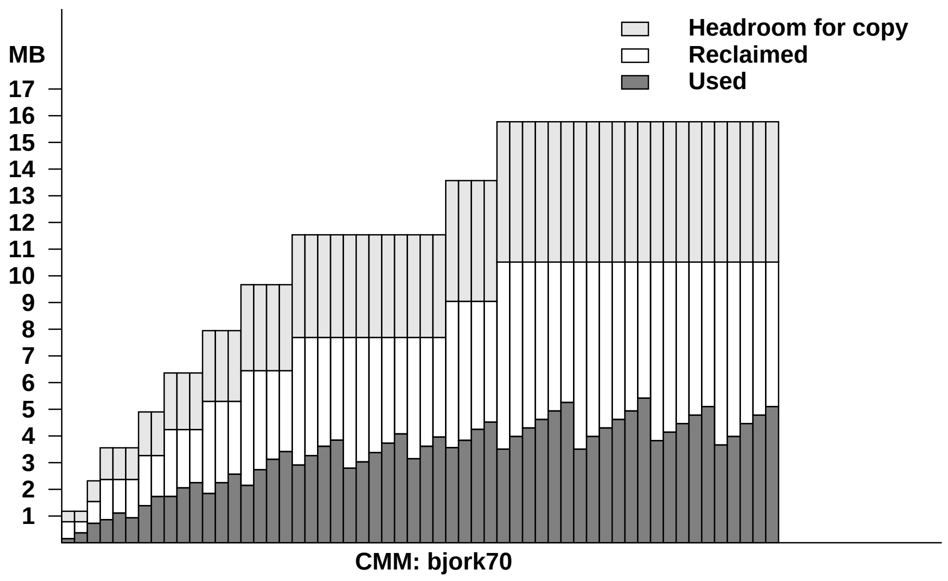

Figure 5: CMM: biork70

All these benchmarks were run on machines with enough physical memory to avoid swapping, since the time spent in swapping is difficult to measure: it is not reported by the Unix system call `getrusage()` and may vary at each execution.

13.2 Opportunistic Collection

The idea of using a temporary area for allocating temporary objects in the Buchberger algorithm was suggested by Faugère [23] who realised it through a manual allocator/deallocator in his system. The CMM generalises this idea and automates the mechanism.

One may question however whether other known techniques would work as well. For instance one may think that an incremental collector would provide similar benefits, since it could recover quickly temporary objects.

To verify this hypothesis, we tested our algorithm also using the incremental/generational version of Böhm collector.

For comparison reasons Table 4 reports the performance data in the configuration used in the tests in this section: interior pointer recognition disabled, `BigInt` objects allocated as atomic.

Table 4: Böhm-NIP

Bench	**User time**	**System time**	**Memory**
katsura6	240.64	0.32	2584
bjork70	623.47	0.85	8400
katsura7	5914.84	8.29	14656

The first three columns in Table 5 show the results for the incremental version of Böhm collector.

In this configuration there is a decrease in performance, both in terms of memory and time, with respect

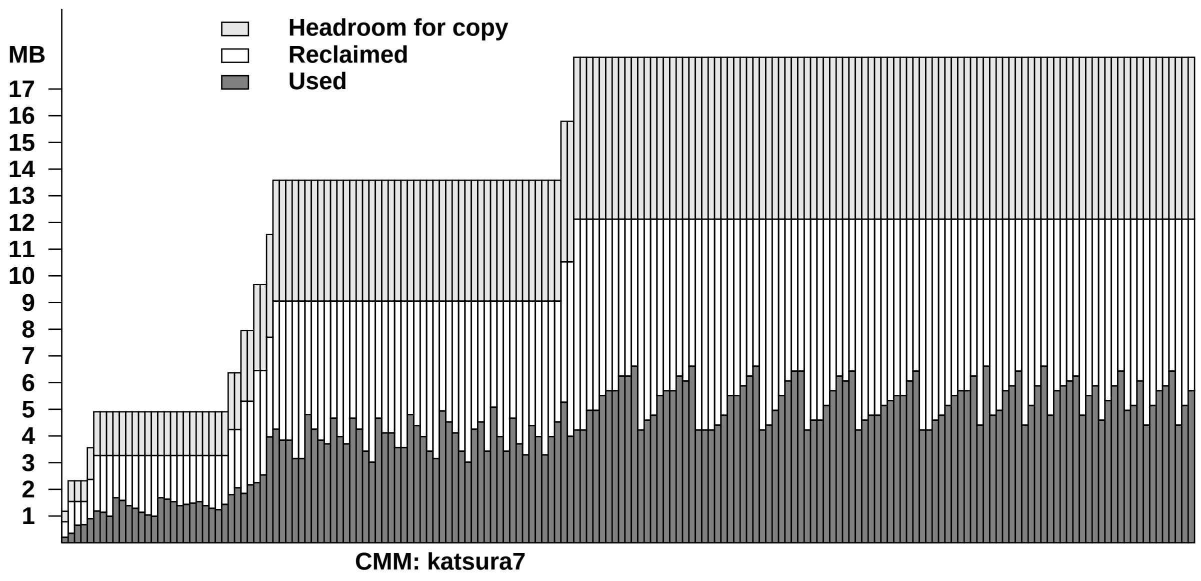

Figure 6: CMM: katsura7

to the standard version of Böhm collector. This can be explained by the extra work required to maintain the information for pages and by the costs of traps into the operating systems, which result from implementing the *write barrier* by write protecting virtual memory pages: note in fact the significant increase in system time.

The next experiment is to consider whether "opportunistic" scheduling of collection can be beneficial, i.e. invoking the collector at the appropriate time, when little live data is present. We tried emulating this technique manually, exploiting a hook provided in the Böhm collector through function `GC_maybe_gc()` that tests whether a collection is in order and then performs it. We added an explicit call in the body of `simplify()`, similarly to what we did for using the `TempHeap`.

The results in the second group of columns in Table 5 show that this can be effective: if we trace the number of calls with the profiler, we notice that only 4 of the 92 effective collections required in the computation are triggered automatically, confirming that the garbage collector is invoked at the right time. However this solution leads only to slight improvements, probably because the Böhm collector still needs to scan the whole root set: the collector for the CMM `TempHeap` instead limits itself to consider its roots, which are just the `CurrentPolynomial` and `CurrentMonomial`. No difference is noticeable in the overall amount of memory used since the collector obtains memory in chunks from the operating system.

Finally we examined the effects of calling explicitly the collector opportunistically within the incremental/generational version of the collector, again by calling at the most suitable time the function `GC_maybe_gc()`. We are grateful to Hans Böhm who made suitable changes to get this feature to work in the latest release of his collector.

The results appear in the last columns in Table 5 and show no relevant improvement. This is probably due to the fact that incremental collection triggers automatically too often, before it is time for the opportunistic invocation.

13.3 Summary

The Böhm collector overall showed very good performance in all our tests, however also with this collector we had to tune the application for its use in order to avoid thrashing or performance degradation. In Böhm collector user intervention can be in the form of either disabling interior pointer recognition or allocating

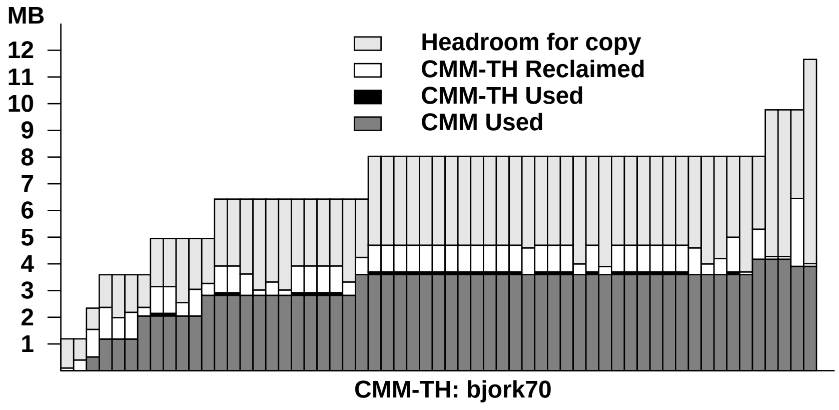

Figure 7: CMM-TH: bjork70

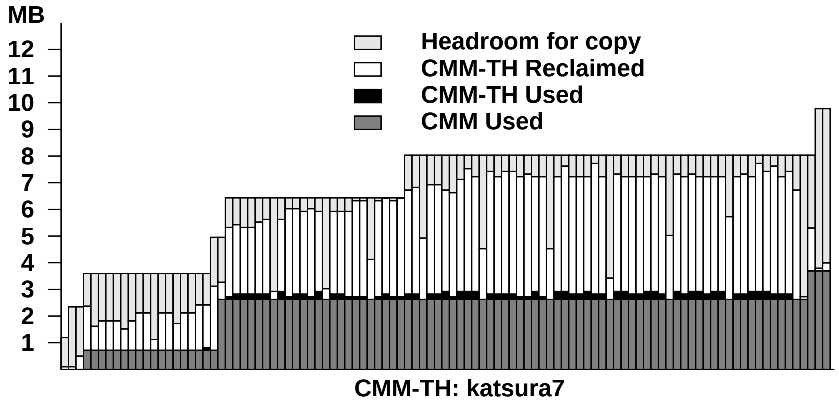

Figure 8: CMM-TH: katsura7

specially objects that do not contain pointers.

The first option must be used carefully since it is pervasive and unsafe. The second option is the most effective but requires a more detailed analysis and hints from the user. User intervention is required quite similarly to what is needed when using CMM.

Therefore to properly compare the CMM and Böhm collectors, we should consider the latter configured with interior pointer recognition and user selection of the allocation routine for objects without pointers.

Table 6 summarises this comparison.

We can conclude that in similar configurations CMM and Böhm collector have similar performance (remember that the actual use of memory for CMM is approximately 2/3 of the maximum reported here).

Since the techniques and implementation are quite different, one may argue that the performance they achieve is close to the best achievable with conservative collectors on stock hardware without compiler support.

By exploiting customisation however, CMM can further improve on this performance, as our experience with the `TempHeap` demonstrates. The results of the `TempHeap` could not be achieved by other simple means like incremental or generational collection or by simulating an opportunistic collector.

In any case, some form of user intervention in customising or supplying hints to a collector appears to be necessary to achieve the best performance (or in some cases just termination) in memory intensive applications like computer algebra.

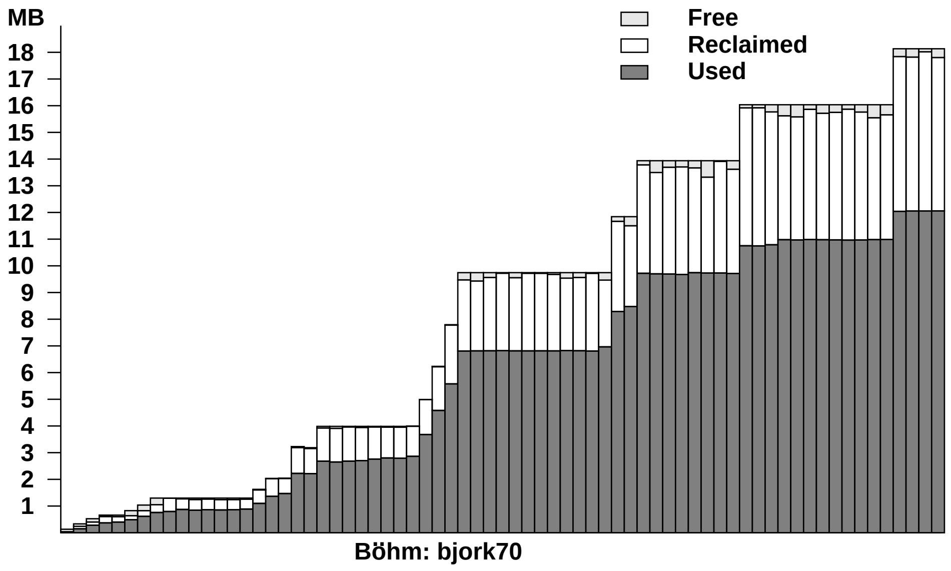

Figure 9: Böhm: bjork70

Table 5: Böhm-NIP incremental, opportunistic and both

	Incremental			Opportunistic			Incremental/Opportunistic		
Bench	**User time**	**System time**	**Memory**	**User time**	**System time**	**Memory**	**User time**	**System time**	**Memory**
katsura6	271.15	35.55	2584	227.90	0.37	2584	227.80	0.17	2584
bjork70	896.69	104.49	8336	618.72	0.90	8336	660.65	1.25	8336
katsura7	5943.87	525.59	16768	5731.73	5.14	14656	6643.49	3.78	14656

14 Assessment

Having chosen to base the design of the collector on inheritance and specialisation proved to be convenient to achieve an open design that can be easily extended. But what are the drawbacks of such choice?

14.1 Space Overhead

Collected objects must inherit from class `CmmObject`, which declares `traverse()` as a virtual function and therefore space overhead is added to each object. Some overhead is however inevitable to enable garbage collection. Other solutions either add one word of header to identify the type of objects in the heap or allocate objects in a separate region for each type, which also cause some waste of memory. On the other hand we were able to avoid any space overhead except what C++ needs for implementing objects with virtual functions and 2 bits per word in global tables.

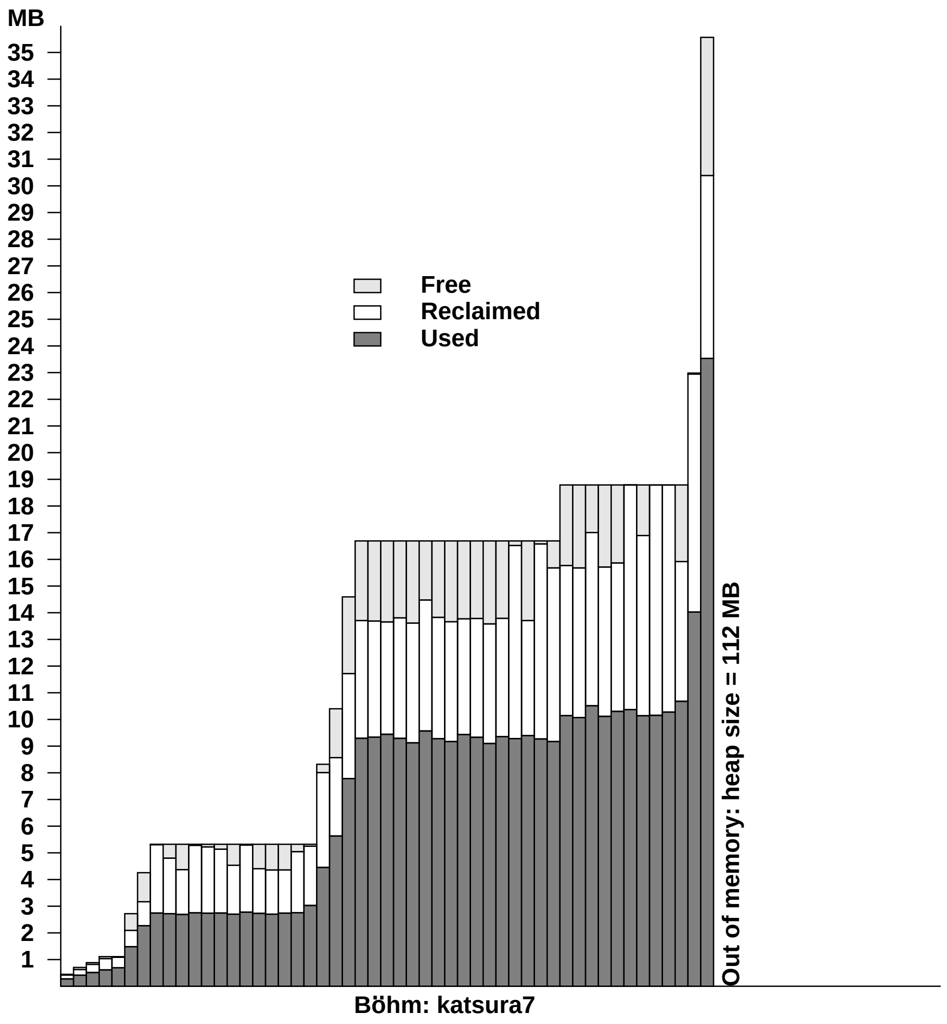

Figure 10: Böhm: katsura7

14.2 Overloading `new`

The CMM exploits the placement syntax of the `new` operator to determine where and how to allocate an object. This of course limits user code from using this feature, even though just for collected objects. It is hard to assess how bad is this limitation: in fact Ellis and Detlefs [22] argue that there is no reason to overload `new` for a collected class, whose allocation is performed by the garbage collector methods. On the other hand, in the CMM, class `CmmObject` is just a class defined in a public library whose code is accessible, so there is no difficulty in defining derived classes from it, with suitable specialisation of its `new` operator.

One case in which we found this useful was to define collectable objects of variable size.

Consider the following example:

```
class BigNum {
  BigNum(int l) {
    length = l;
    limbs = new int[l];
  }
  int length;
  int* limbs;
```

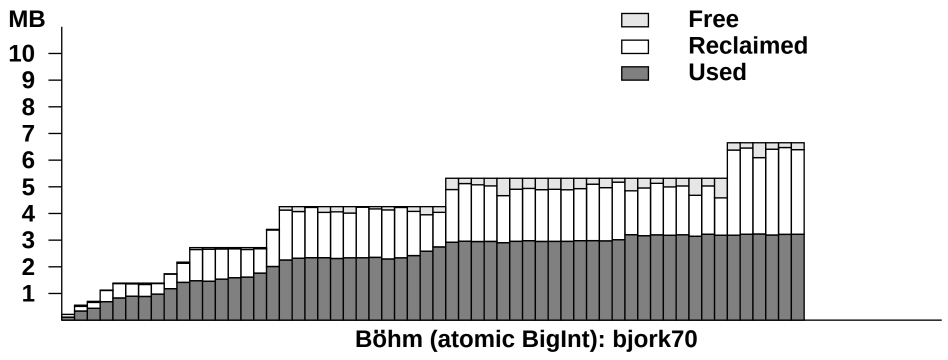

Figure 11: Böhm (atomic BigInt): bjork70

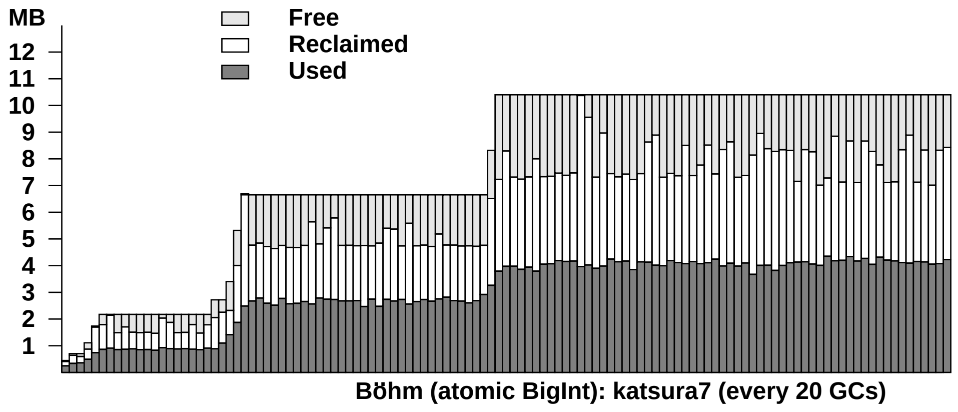

Figure 12: Böhm (atomic BigInt): katsura7 (every 20GCs)

```
}
```

With this definition of `BigNum`, creating a `BigNum` will require two memory allocations and accessing the array `limbs` will require an extra pointer indirection.

To improve this solution, CMM provides the class `CmmVarObject`, a derived class from `CmmObject`, which provides a `new` operator with an additional parameter for the size of the variable member. The following example illustrates its use:

```
class BigNum : public CmmVarObject
{
  int length;
  int limbs[1];  //  size determined at object creation
};

  BigNum* num = new(256 * sizeof(int)) BigNum;
```

The object `num` is created in the default heap and has room for 256 integers. The implementation of class `CmmVarObject` might be however compiler dependent.

Table 6: Comparison CMM - Böhm (atomic BigInt)

Bench	**CMM sec.**	**Böhm sec.**	**CMM Kbyte**	**Böhm Kbyte**
katsura6	263.14	235.47	3051	2228
bjork70	700.40	685.42	12979	7156
katsura7	5749.08	5560.16	17519	11008

14.3 Applicability

The technique of CMM is applicable to other languages, even non object-oriented languages and C in particular, by using the mechanism of *callbacks* (or *upcalls*) to customise the collector, as in the original Bartlett's implementation or in the Portable Common Runtime [40]. A user would register a specific callback routine with the general garbage collector, for use on a specific type of objects. So when the garbage collector recognises one of these objects during traversal, it applies the appropriate callback to collect the object. This would allow us to produce a CMM library more easily linkable to user code, avoiding linkage problems due to C++ virtual functions.

In the Portable Common Runtime [40], when an upcall is registered with the collector, a type code is returned that can be used to tag objects at their creation. This is used to support different programming languages, which have different allocation primitives.

Another possibility would be to use the technique of Run Time Type Description [28] in order to obtain the layout information for objects needed by the collector to perform tracing.

14.4 Standard Template Library

The C++ Standard Template Library (STL) [37] provides basic data structures and algorithms to support the fundamental building blocks for general programming. The library is structured in several components, including: algorithms – define a computational procedure; containers (e.g. vector, list, deque, set) – manage a set of memory locations; iterators – provide a means for an algorithm to traverse through a container.

STL addresses the issue of memory management by providing a standard set of requirements for allocators, which are objects that encapsulate information about a particular memory model. All of the containers in STL are parametrised in terms of allocators.

Memory management can be customised, by supplying an allocator class in the type definition of an object. For instance:

```
deque<int, MyAllocator> myQueue(100);
```

creates a queue of 100 integers, allocated through class `MyAllocator`.

It would be possible to use CMM to define an allocator complying with the requirements of STL. However the original STL approach to memory management is to associate the storage discipline to the type of an object. This would mean for instance that polynomials allocated in one heap would be of a different type as polynomials allocated in a different heap. This does not allow the allocator policy to be determined at runtime and is quite in contrast with the CMM philosophy that the allocation policy is chosen at creation time rather than definition time.

The STL<ToolKit> by ObjectSpace Inc. [32] allows allocators to be assigned on a per-object basis, by supplying an allocator parameter for object creation:

```
CmmAllocator alloc;               //  CMM-based allocator
deque<int> myQueue (100, alloc); //  Allocate storage from CMM
```

Based on this implementation of STL one can derive a set of container classes augmented with automatic memory management by CMM. This entails deriving such classes from class `CmmObject` and supplying the appropriate method `traverse()` for them.

15 Related Work

We already mentioned the Böhm-Weiser collector [10], a well known collector for C, which is fully conservative and uses the technique of mark-and-sweep. It can handle specially atomic objects (containing no pointers) and it can optionally support incremental and generational collection.

Work on adding garbage collection to C++ has been done by D. Samples and by D. Edelson. Samples [36] proposes modifying C++, to include a garbage collection environment as part of the language. This may be a good long term approach for garbage collection in C++ but does not solve the immediate needs for portable garbage collection of present applications. We believe that our work demonstrates how the flexibility of object oriented languages can be used to implement a complex environment, like CMM, without requiring modifications to the language.

Edelson [20] has been experimenting with the coexistence of different garbage collection techniques. The flexibility of the solutions he adopts in his approach allows the coexistence of different garbage collectors, but he does not provide any interface to the user to customise and/or define his own memory management facilities.

Detlefs [17] proposes the use of "smart pointers" template classes as an interface for the use of garbage collection in C++. Such smart pointers also support run-time type queries, giving the program the ability to ascertain the allocated type of an object at run time. The interface is designed as a framework that could support a number of different collection algorithms. While the proposed implementation as a C++ library is not the most efficient, Detlefs argues about the importance of standardising the interface.

Ellis and Detlefs [22] propose some extensions to the C++ language to allow for collectable objects. The major change is the addition of the type specifier `gc` to specify which heap to use in allocating the object or a class. They also propose to change the operator `new T` to call the collector allocator when `T` is a `gc` type, and as a consequence of this, the overloading of `new` and `delete` operators for `gc` classes is forbidden. While the `gc` keyword is compatible with our solution of inheriting from the base class `CmmObject`, the constraint on `new` needs to be relaxed to allow overloading of `new` when additional arguments are present. Otherwise this constraint will block the possibility of using different heaps for the same kind of objects in different portions of a program. Other suggestions from the Ellis-Detlefs proposals are quite valuable, for instance making the compiler aware of the garbage collection presence and avoid producing code where a pointer to an object (which may be the last one) is overwritten. This can happen for instance in optimizing code for accessing structure members. To reduce the cost of traversing conservatively the whole uncollected heap, Ellis and Detlefs propose using the same synchronization technique used for generational collection: all pages in the uncollected heap are write-protected, and the subsequent write-protection fault caused by the program tell the collector which pages have been modified. Only these pages will have to be scanned for roots.

Vo's allocator [38] allows organising memory into different "regions", each managed with a different discipline. A program can select which region to use either at link time or within different parts of the application. This supports experimentation with different allocators to tune memory management to specific applications or to perform debugging and profiling. Customisation of the allocator is achieved by supplying specialised functions to get raw memory and to manage allocation. Of course Vo's allocator does not perform automatic memory reclamation.

The basic design of CMM was presented in [2, 3]: the major differences with the present work are in a new version of the collector algorithm to limit the cases of recursion, statistics on its benefits, support for arrays of `CmmObjects`, support for virtual base classes and references in classes, the extension of the framework with the mark-and-sweep heap, detailed description on how to extend the framework.

In [4] preliminary results of benchmarking with CMM were presented.

16 Conclusion

The CMM offers garbage collection facilities which can be taylored to the need of an application. Programmers can use a generic collector, a specific collector or no collector at all, according to the need of each algorithm. The algorithm can be in control, when necessary, of its memory requirements and does not have to adapt to a fixed memory management policy.

Even though limited to a class of applications, the benchmarks presented show that CMM performs generally better than the default configuration of the Böhm collector. This is not surprising since CMM exploits information on the layout of pointers within objects. If some information of this kind is made available to Böhm collector, for instance by using a different allocator for objects without pointers, the performance of the two collectors become similar. This confirms the importance of supplying layout information for achieving the best collector performance.

By exploiting customisation however, CMM can further improve on this performance, particularly by allowing the programmer to taylor the collector strategy to the pattern of memory use in his application. We have described one example of this customisation in the case of one particularly demanding algorithm for computer algebra. The customisation is however general enough to be applicable in several other situations.

The level of performance achieved through customisation does not appear to be achievable directly with other means, like using incremental, generational or opportunistic collectors.

However determining where and which technique to use might not be obvious and may require some experimentation and benchmarking. Our experiments showed that slight variations on an algorithm could have big impacts hard to predict (as in the case of the Boehm collector with/without interior pointer detection).

The flexibility and user control provided by CMM make such experimentations possible and allow tackling cases that are dealt poorly by general purpose conservative collectors.

The CMM is implemented as a C++ library, produced after a complete redesign from the original Bartlett's code. It is being heavily used in the implementation of high demanding computer algebra algorithms in the PoSSo project. The CMM provides the required flexibility with good performance as compared to versions of the same algorithms performing manual allocation.

17 Availability

The sources for CMM are available for anonymous ftp from site `ftp.di.unipi.it` in the directory `/pub/project/posso`. The CMM code may be used, modified and copied under minimal copyright restrictions. Comments, suggestions and bug reports should be addressed to `cmm@di.unipi.it`.

18 Acknowledgements

Carlo Traverso and John Abbott participated to the early stages of the design. J.C. Faugère suggested the idea of a temporary heap for the Buchberger algorithm. Joachim Hollman and Fabrice Rouillier helped in testing various releases of the implementation. Discussions with J. Ellis were useful to ensure compatibility of his proposal with our framework. The anonymous referees provided valuable suggestions to improve the content and presentation. Christian Heckler provided critical comments on the paper and CMM documentation and supplied some examples and insights to fix some bugs in the code. Paul Wilson helped clarify some issues on opportunistic collectors. With Hans Böhm we discussed the performance data and the behaviour of his collector.

References

[1] K. Arnold, J. Gosling, 'The Java Programming Language', Addison Wesley, (1996).

[2] G. Attardi and T. Flagella, 'A customisable memory management framework', *USENIX C++ Conference Proceedings*, Cambridge, Massachusetts, 123–142 (1994).

[3] G. Attardi and T. Flagella, 'Customising object allocation', in M. Tokoro and R. Pareschi (eds.) *Object-Oriented Programming, Proceedings of the 8th ECOOP, Lecture Notes in Computer Science* **821**. Berlin:Springer-Verlag, 320–343 (1994).

[4] G. Attardi, T. Flagella and P. Iglio, 'Performance Tuning in a Customizable Collector', *Memory Management*, IWMM 95, *Lecture Notes in Computer Science* n. 986, Springer-Verlag, Berlin, 179–196 (1995).

[5] H. G. Baker Jr., 'The Treadmill: Real-time garbage collection without motion sickness', *ACM SIGPLAN Notices*, **27**(3), 66–70 (1992).

[6] H. G. Baker Jr., 'Personal communication', (1996).

[7] J. F. Bartlett, 'Compacting garbage collection with ambiguous roots' Tech. Rep. 88/2, DEC Western Research Laboratory, Palo Alto, California, February 1988.

[8] J. F. Bartlett, 'Mostly-copying collection picks up generations and C++', Tech. Rep. TN-12, DEC Western Research Laboratory, Palo Alto, California, (1989).

[9] P. Beard, 'Personal communication', (1997).

[10] H. J. Böhm and M. Weiser, 'Garbage collection in an uncooperative environment', *Software Practice and Experience*, **18**(9), 807–820 (1988).

[11] H. J. Böhm, 'Space Efficient Conservative Garbage Collection', in *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, *ACM SIGPLAN Notices*, **28**(6), 197–206, (1993).

[12] H. J. Böhm, 'Personal communication', (1996).

[13] B. Buchberger, 'Gröbner bases: an algorithmic method in polynomial ideal theory', *Recent trends in multidimensional systems theory*, N. K. Bose, ed., D. Reidel Publ. Comp., 184–232 (1985).

[14] C. J. Cheney, 'An nonrecursive list compacting algorithm', *Communications of the ACM*, **13**(11), 677–678 (1970).

[15] G. O. Collins, 'Experience in automatic storage allocation', *Communications of the ACM*, **4**(10), 436–440, (1961).

[16] D. L. Detlefs, 'Concurrent garbage collection for C++', CMU-CS-90-119, School of Computer Science, Carnegie Mellon University, (1990).

[17] D. L. Detlefs, 'Garbage collection and run-time typing as a C++ library', in *Proceedings of the 1992 Usenix C++ Conference*, (1992).

[18] D. L. Detlefs, Al Dosser and B. Zorn, 'Memory allocation costs in large C and C++ programs', *Software Practice and Experience*, **24**(6), 527–542 (1994).

[19] 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, (1978).

[20] D. R. Edelson, 'Precompiling C++ for garbage collection', in *Memory Management*, Y. Bekkers and J. Cohen (Eds.), Lecture Notes in Computer Science, n. 637, Springer-Verlag, 299–314 (1992).

[21] D. R. Edelson, 'A mark-and-sweep collector for C++', *Proc. of ACM Conference on Principle of Programming Languages*, (1992).

[22] J. R. Ellis and D. L. Detlefs, 'Safe, efficient garbage collection for C++', Xerox PARC report CSL-93-4, (1993).

[23] J. C. Faugère, 'Résolution des systèmes d'équations algébriques', PhD thesis, Université Paris 6, (1994).

[24] T. Goldstein, 'Personal Communication', Sun Microsystems, (1994).

[25] D. R. Hanson, 'Fast allocation and deallocation of memory based on object lifetimes', *Software Practice and Experience*, **20**(1), (1990).

[26] Christian Heckler, 'Personal communication', University of Paderborn, (1997).

[27] J. D. Ichbiah et al., 'Rationale for the design of the ADA programming language', *ACM SIGPLAN Notices*, **14**(6), (1979).

[28] S. V. Kakkad, M. S. Johnstone, and P. R. Wilson, 'Portable Runtime Type Description for Conventional Compilers', submitted to USENIX '97, (1996).

[29] S. Katsura et al., 'Distribution of effective field in the Ising spin glass of the $\pm J$ model at $T = 0$', *Cell Biophysics* **11**, 309–319 (1987).

[30] D. Moon, 'Garbage collection in a large Lisp system', in *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, ACM Press, 235–246, (1984).

[31] G. Nelson, editor, *Systems Programming with Modula3*, Prentice Hall, (1991).

[32] 'ObjectSpace Inc., STL<ToolKit>, http://www.objectspace.com.

[33] D. Plainfossé and M. Shapiro, 'A Survey of Distributed Garbage Collection Techniques', *Memory Management*, IWMM 95, *Lecture Notes in Computer Science* n. 986, Springer-Verlag, Berlin, 211–250, (1985).

[34] J. Richter, 'Advanced Windows: the developer's guide to the Win32 API for Windows NT and Windows 95', Microsoft Press, (1995).

[35] F. Rouillier, 'Personal communication', (1994).

[36] A. D. Samples, 'GC-cooperative C++', *Lecture Notes in Computer Science*, **637**, Springer-Verlag, 315–329 (1992).

[37] A. Stepanov and M. Lee, 'The Standard Template Library', Hewlett-Packard Company, (1995).

[38] K.-P. Vo, 'Vmalloc: a general and efficient memory allocator', *Software Practice and Experience*, **26**(3), 357–374 (1996).

[39] D. Weinreb, D. Moon and R. M. Stallman, 'Lisp Machine Manual' Massachusetts Institute of Technology, Cambridge, Massachusetts, (1983).

[40] M. Weiser, A. J. Demers, C. Hauser, 'The Portable Common Runtime Approach to Interoperability', Proceedings of the Twelfth ACM Symposium on Operating System Principles, ACM Press, 114–122 (1989).

[41] E. P. Wentworth, 'Pitfalls of conservative garbage collection', *Software Practice and Experience*, **20**(7), 719–727 (1990).

[42] P. R. Wilson and T. G. Moher, 'Design of the Opportunistic Garbage Collector', Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA) '89 Proceedings, ACM Press, 23–35 (1989).

[43] P. R. Wilson, 'Uniprocessor garbage collection techniques', in *Memory Management*, Y. Bekkers and J. Cohen (Eds.),*Lecture Notes in Computer Science*, **637**, Springer-Verlag, 1–42 (1992).

[44] P. R. Wilson et al., 'Dynamic Storage Allocation: A survey and Critical Review', *Memory Management*, IWMM 95, *Lecture Notes in Computer Science*, **986**, Springer-Verlag, Berlin, 1–116 (1995).

[45] P. R. Wilson et al., 'Memory allocation policies reconsidered', Technical report, University of Texas at Austin, Department of Computer science, (1995).