A New Garbage Collector for XEmacs

Diplomarbeit

Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Arbeitsbereich für Programmiersprachen und Übersetzer

Marcus Crestani

31. August 2005

Betreuer: Prof. Dr. Herbert Klaeren
Dr. Michael Sperber

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Tübingen, 31. August 2005

Dank

Ich danke Mike Sperber für seine hervorragende Betreuung.

Mein herzlichster Dank gilt meinen lieben Eltern und Großeltern, die mir dieses Studium ermöglicht haben. Außerdem danke ich meiner Freundin Bernadette für ihre Unterstützung und Motivation.

Zusammenfassung

XEmacs braucht eine schnelle und erweiterbare automatische Speicherverwaltung.

Der Garbage-Collector des XEmacs ist langsam: Der Benutzer wird bei seiner Arbeit durch lange Programmpausen unterbrochen, die der Garbage-Collector verursacht. Da zudem der Speichermanager eng mit dem restlichen Programmcode verbunden ist, gibt es keine übersichtliche Schnittstelle zwischen XEmacs und der Speicherverwaltung. Dies erschwert die Überarbeitung des Speichermanagers. Mein Ziel ist es, die Programmpausen zu verkürzen und den Speichermanager zu modularisieren.

Deshalb verbessere ich den Garbage-Collector: Ich beschreibe einen inkrementellen Garbage-Collector für XEmacs, der die Unterbrechungen durch die Garbage-Collection verkürzt, indem das Programm abwechselnd mit dem Garbage-Collector ausgeführt wird. Die dafür erforderliche Schreibbarriere beschreibe ich ausführlich.

Im Allgemeinen bereite ich XEmacs darauf vor, andere Kollektoren nutzen zu können. Dafür modularisiere ich den Speichermanager, beseitige Abhängigkeiten und definiere eine einfache und übersichtliche Schnittstelle.

Abstract

XEmacs needs a fast and extensible automatic memory manager.

XEmacs's garbage collector is slow: The user's work is interrupted by the garbage collector, which is causing long and annoying pause times. Additionally, the memory manager is tightly coupled with other parts of XEmacs, it lacks a straightforward interface and is not easily improvable. My intention is to decrease pause times and modularize the memory manager.

Therefore, I improve the garbage collector: I describe an incremental garbage collector for XEmacs that keeps garbage collection pause times short by interleaving small amounts of collection work with program execution. I introduce the needed write barrier algorithms in detail.

More generally, I enable XEmacs to use alternative garbage collection schemes by modularizing the memory manager, removing dependencies, and defining a straightforward interface.

Contents

List of Figures

Chapter 1

Introduction

Garbage collection is the automatic reclamation of computer storage [Wil92]. A *garbage collector* is a program that performs automatic reclamation: it has to find unused objects and make their space available for reuse again.

XEmacs is a powerful, highly customizable text editor and development environment [WTB+]. XEmacs's abilities exceed simple text insertion or deletion; it can indent and color source code of different programming languages automatically, view two or more files at once, and it can be used as a web browser and email reader.

XEmacs is programmable: The user can add new commands and implement complex applications entirely within the system. Therefore, XEmacs comes with its own programming language, *Emacs Lisp*. Big parts of XEmacs and all of its extensions are written in this dialect of Lisp.

Like other Lisp-style programming languages, Emacs Lisp comes with an automatic memory manager, a garbage collector. Basic functionality of XEmacs is written in C, as is the memory manager.

1.1 Motivation

XEmacs has grown over the years; its automatic memory manager was written in the 1970's. The user has to deal with the outdated memory manager continually: When it runs, the system freezes for several seconds, XEmacs does not react to keystrokes, and the display is not updated.

Even the developers' work is affected by the memory manager: its source code is complex, and the memory manager is tightly coupled to the rest of XEmacs. Simple modifications to XEmacs may lead to vast modifications to the memory manager and vice versa. The memory manager lacks a straightforward interface and is not easily changed. The developer has to cope with many details and cannot concentrate on the intended work.

Purpose of my diploma thesis is to replace the old garbage collector with a new one: In this document I describe an incremental garbage collector for XEmacs that keeps garbage collection pause times short by interleaving small amounts of collection work with program execution. Additionally, I modularize the memory manager to make it amenable to future improvements.

In chapter 2, I give an overview about automatic memory management in

general and incremental garbage collection techniques in detail. Chapter 3 describes the implementation of a write barrier needed for an incremental garbage collector and further changes to XEmacs. Chapter 4 lists the results of several performance and memory usage measurements along with an evaluation. The conclusion and future work can be found in chapter 5.

This work builds on earlier work done in the context of my term paper [Cre04] that describes changes to the allocator that are required by the new garbage collector.

Chapter 2

Overview

This chapter provides an overview about automatic memory management in general and incremental garbage collection techniques in detail. A garbage collector is part of an *automatic memory manager*, whose basics are described in the first part of this chapter. The second part of this chapter focuses on an improved garbage collection algorithm.

2.1 Automatic Memory Management

A program with automatic memory management logically consists of two parts [JL96]:

1. The *client*—the part of the program which does "useful" work—requests memory for its objects.

2. The *memory manager* serves the client's memory needs and reclaims unused memory automatically.

After the client has allocated memory for its objects, it does not need to free the memory occupied by an object explicitly when it is no longer in use: The memory manager automatically determines which objects the client no longer uses and frees these objects. A memory manager consists of two parts: the *allocator* and the *garbage collector*. Figure 2.1 shows the basic structure of a program with automatic memory management.

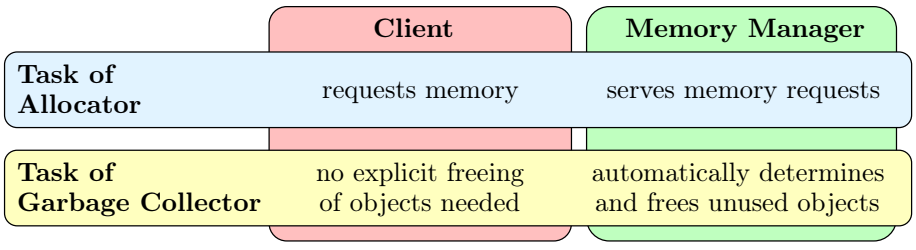

Figure 2.1: Structure of a program with automatic memory management

2.1.1 The XEmacs Allocator

The memory administrated by the allocator is called the *heap*. The allocator, to quickly find successive blocks of free memory of a certain size, or to coalesce adjacent free blocks into one big area, uses a data structure called the *free list* that represents unused memory in a linked list. The allocator traverses the free list to find a sufficiently-sized slot for newly allocated objects.

In this document I concentrate on the garbage collector—for a more thorough treatment of the allocator, see my term paper [Cre04].

2.1.2 The XEmacs Garbage Collector

The garbage collector frees objects that can no longer be accessed by the running program. These objects are referred to as *garbage* or *dead*. Objects that are being used by the client are called *live*.

The process of garbage collection consists of two parts [Wil92]:

1. *Garbage detection*: distinguishing live objects from garbage
2. *Garbage reclamation*: reclaiming the storage occupied by garbage objects for later use by the client

XEmacs is using one of the earliest garbage collection algorithms: the *mark-and-sweep* garbage collection. It is named for the two phases that implement the garbage collection process described above [JL96, Wil92]:

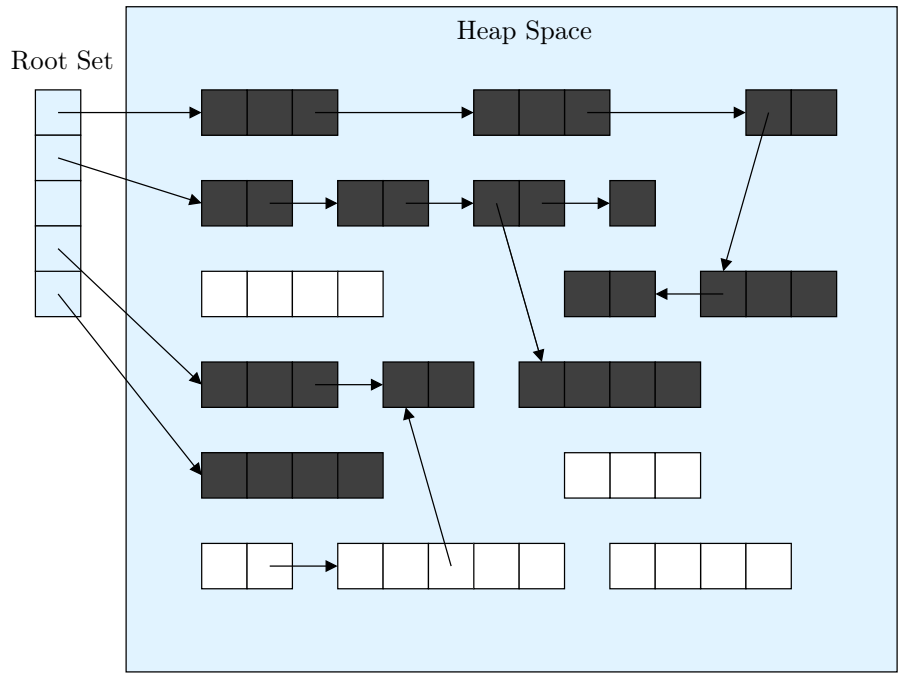

Figure 2.2: Mark-and-sweep garbage collection after the mark phase has completed

1. The *mark phase* distinguishes live objects from garbage by constructing a *reachability graph*. This is done by a global traversal of all live objects, starting at the *root set*. The root set contains the pointers to the initially reachable objects. Typically, this includes the global variables and the local variables on the activation stack. The client has to register the root set with the memory manager.

   The traversal follows all pointers from the root set transitively to other objects on the heap. Any object reachable from a live object is also live. Thus the set of live objects is simply the set of objects on any directed path of pointers from the roots. For all traversed objects the garbage collector sets a bit—the objects are *marked*.

   The client is often called *mutator*, since, as far as the collector is concerned, its only task is to change or mutate the connectivity of the active objects on the heap.

2. The *sweep phase* frees dead objects: The entire heap is scanned, then all unmarked objects are *swept*, and their space is reclaimed.

Figure 2.2 shows a heap directly after the mark phase is finished. The objects colored black are reachable from the root set and therefore marked. The white objects are garbage, and are swept in the next phase.

Traditional mark-and-sweep garbage collectors have two major problems:

1. Management of free memory is difficult because the memory of a running program becomes fragmented: Reclamation of objects leads to holes within the allocated memory. This leads to the accumulation of small regions of free storage that are too small to be useful for allocation, even though the sum of free space may be more than sufficient.

2. The cost of a collection is proportional to the size of the heap, including both live and garbage objects:

   - All live objects must be marked, and
   - all garbage objects must be freed.

The first problem has already been taken care of in my previous work on the allocator [Cre04]. Here, I deal with the second problem: Traditional mark-and-sweep garbage collectors suspend the client during their work. Large heaps with many objects and many references and large amounts of garbage lead to time-consuming mark and sweep phases. The client is suspended for a long time, which is annoying in interactive or real-time applications.

Consequently, it is necessary to take a closer look to XEmacs's traditional mark-and-sweep algorithm to determine which phase of the mark-and-sweep scheme causes the long pause times.

2.1.3 XEmacs's Traditional Mark-and-Sweep Algorithm

This section provides results of measurements of the execution time XEmacs's traditional mark-and-sweep garbage collection is spending in its phases.

It is difficult to measure the performance of garbage collection algorithms [JL96]. The execution time of a full collection cycle depends largely on the

topology and size of the data in the heap. Even simple issues, such as minor changes to the size of the heap or the layout of objects, can cause radically different results. The topology and size of its heap objects is defined by the way XEmacs is used. Therefore, I assume a set of standard XEmacs applications and user behavior patterns to be able to measure the collector's performance with these patterns. This *standard XEmacs usage pattern* is described in more detail in section 4.3.1.

The measurements of time XEmacs is spending in the two phases of the traditional mark-and-sweep scheme produce the following results:

- share of time spent in mark phase: 60%–80%
- share of time spent in sweep phase: 1%–15%

The broad derivation of the resulting values are caused by the difficulties described above of making objective performance measurements. With a standard XEmacs usage pattern, the mark phase consumes most of the garbage collection times.

Of course, it is easy to write an XEmacs application that produces significantly more garbage than live objects: If an application generates a large number of objects that immediately become garbage, these objects are not traversed during the mark phase, but the collector needs to free all these garbage objects during sweep phase. This would lead to an inverted result in the above measurements, since more objects have to be swept. But this is not usually the way the memory layout evolves, assuming an average XEmacs usage.

Hence, the mark phase of its traditional mark-and-sweep garbage collection causes long pause times that interrupts user interaction and the program's reactivity. The next section discusses improvements to the mark-and-sweep garbage collection.

2.2 Incremental Techniques

When garbage collection is carried out as one atomic action while the program is halted, the pause times are long. To shorten the pause times, small units of garbage collection must be interleaved with small units of program execution. This *incremental* garbage collection allows the running program to be able to react to user interaction and resume whatever task it may have pending while garbage collection is in progress.

The difficulty with incremental techniques is that, while the collector traverses the graph of reachable objects, the graph may change, as the running client may mutate the graph while the collector "is not looking" [Wil92]. An incremental scheme must have some way of keeping track of the changes to the reachability graph made by the client behind its back.

The following sections specifies that problems that may occur during incremental traversal of the living objects and describe techniques for dealing with them.

First of all, interruption and later resumption of a garbage collection is costly: it incurs time and memory overhead, which is discussed more precisely later. As shown in section 2.1.3, the mark phase is responsible for the long pause times. Hence, I focus on an incremental mark phase, also called *incremental traversal*.

An incremental sweep phase would not be too difficult to implement, but it requires modifications to the allocator to keep the state of the sweep progress. An experimental implementation has shown that it currently does not lead to noticeable improvements.

The next section introduces an algorithm for incremental traversal.

2.2.1 Tricolor Marking

For understanding incremental garbage collection the abstraction of tricolor marking is useful [Wil92]. Garbage collection traverses the graph of reachable objects and colors them. The objects subject to garbage collection are white at the beginning. By the end of the collection, those that will be retained are colored black. When there are no reachable objects left to blacken, the traversal of live data structures is finished. In traditional mark-and-sweep collectors, this coloring is directly implemented by setting mark bits. In figure 2.2, a set mark bit is indicated by coloring the object black.

In an incremental collector, the intermediate state of the traversal is important because of ongoing mutator activity: the mutator cannot be allowed to change things in such way that the collector will fail to find all reachable objects. To understand and prevent such interactions between the mutator and the collector, it is useful to introduce a third color, grey.

Grey objects have been reached by the traversal, but its descendants may not have been. White objects are changed to grey when they are reached by the traversal. Grey objects mark the current state of the traversal: traversal proceeds by processing the grey objects. Processing a grey object means following its outgoing pointers, and coloring it black afterwards.

The traversal starts with the root set. The objects pointed to from the roots are colored grey. In the next step, all the grey objects are scanned and pointers to their offspring are followed. The grey objects are colored black, after all outgoing pointers have been examined. If the traversal reaches an offspring for the first time—it is colored white—then it is colored grey. If an offspring has already been reached—it is already marked grey or black—nothing has to be done. This way the traversal always terminates.

Figure 2.3 illustrates a state of an incremental traversal: The first few steps of the traversal have already been made; the grey objects denote the current state of progress. The collector is finished with all black objects, and knows nothing yet about the white ones.

In a mark-and-sweep garbage collector, the grey objects correspond to the stack or queue of objects used to control the marking traversal, and the black objects are the ones that have been removed from the queue. Objects that have not been reached are colored white.

2.2.2 Coloring Invariant

Intuitively, the traversal proceeds in a wavefront of grey objects that separates the unreached objects, which are colored white, from the already processed black objects. The algorithm described above does not produce any pointers from black objects to white objects.

This leads to an important invariant, called the *coloring invariant*: There is no direct pointer from a black object to a white object. The importance of this

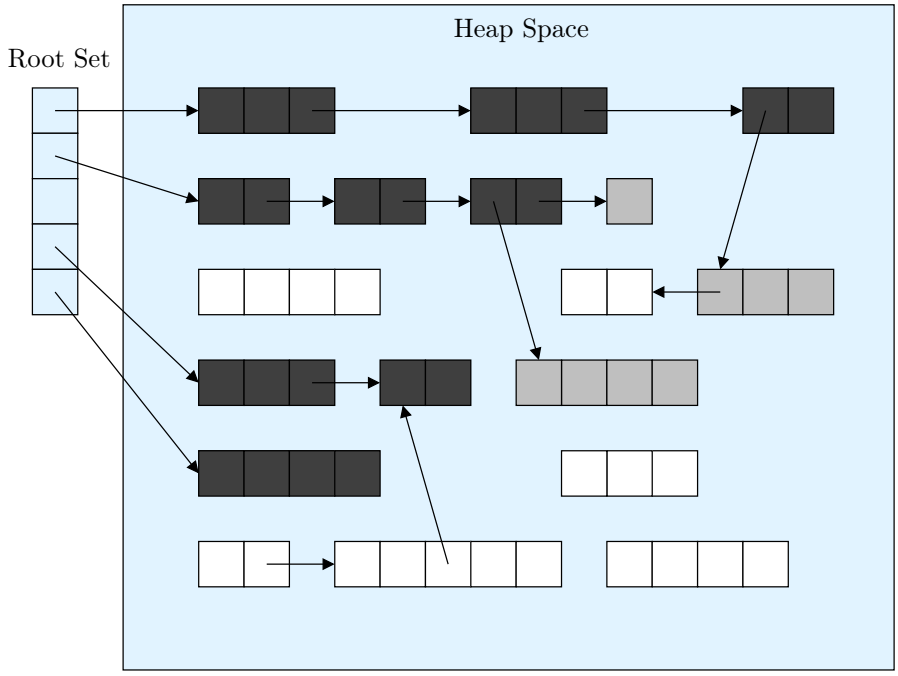

Figure 2.3: State of an incremental traversal

invariant is that the collector must be able to assume that it is finished with black objects, and can continue to traverse grey objects to move the wavefront forward.

When garbage collection is suspended and the mutator resumes, it may violate the coloring invariant if no further provisions are made. When the mutator creates a pointer from a black object to a white one, it must somehow coordinate with the collector, to ensure that the collector's bookkeeping is brought up to date and that the traversal is resumed correctly.

Figures 2.4 demonstrate this need for coordination: Figure 2.4.1 is a cutout from figure 2.3, with letters added to name the objects.

Figure 2.4.1 shows the first modification performed by the mutator. Suppose object A has been completely scanned, and therefore colored black; its descendant B has been reached and colored grey. Object D has also been reached and therefore greyed. The pointer from D to C has not been traversed yet, thus C is still colored white. Now, the mutator writes a pointer to C into A, overwriting the pointer to B. Since the collector assumes that it is already done with A, the newly written pointer to C will not be traversed any more. The client breaks the coloring invariant. For the current scenario, this does not cause the traversal to fail as C still will be reached from D when the traversal is resumed.

In figure 2.4.3 the mutator deletes the pointer from D to C. Now there is no way for object C to be reached by the traversal. If the collector resumes the traversal without any coordination, object C will not be marked. It will be swept and leaves a dangling pointer from object A.

To have the collector falsely reclaim a live object, a white object must become invisible to the collector but still be reachable by the mutator. Figure 2.4.4

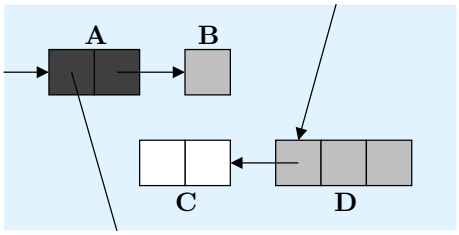

Figure 2.4.1

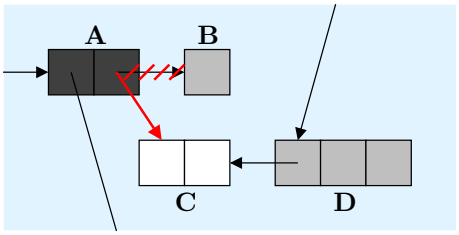

Figure 2.4.2

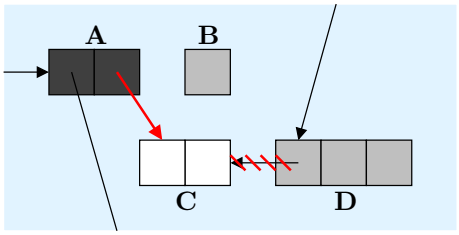

Figure 2.4.3

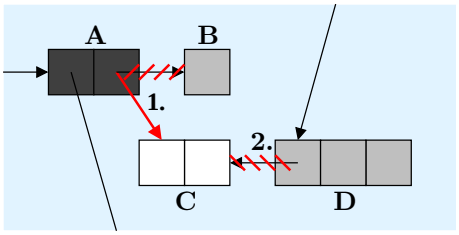

Figure 2.4.4

Figure 2.4: Failing incremental traversal

summarizes the two conditions that would break the traversal:

(Condition 1) Write a pointer to a white object in a black object, and

(Condition 2) remove the original pointer to the white object.

If either of these conditions does not hold, the object will be retained and no special action is required. If Condition 1 does not hold, the graph will not contain any pointers to white objects from black ones. If the white object is reachable, there must be a path to it from a grey object, and the object will be retained; if not, it is garbage and can be freed. On the other hand, if a pointer to a reachable white object is installed in a black object, the white object will still be reached by the collector through the original reference to it if Condition 2 does not hold.

Another important property of incremental collectors illustrates the example in figure 2.4: their degree of conservativeness with respect to the changes made by the mutator during garbage collection. Object B has already been reached by the traversal before its pointer from object A gets overwritten and object B becomes garbage. Since B has already been marked reachable, it will be preserved in this collection cycle, even though it is no longer in use. In general, objects that become garbage during a collection cycle may or may not be reclaimed in this cycle. These objects are called *floating garbage* because the collector has already categorized these objects live before the mutator frees them. Floating garbage is guaranteed to be collected after the next cycle.

The next section shows how the collector and the mutator can be coordinated if the mutator violates the coloring invariant, to prevent traversal failures from happening.

2.2.3 Coordination of Mutator and Collector

There are two basic approaches to coordinating the collector and the mutator: a read barrier or a write barrier. A *read barrier* detects when the mutator attempts to access a pointer to a white object. The read barrier communicates this read access to the garbage collector, which immediately colors the read object grey. This prevents the mutator from writing a pointer to a white object into a black object, because writing a pointer implies seeing it, which then directly colors the object grey. The coloring invariant can no longer be violated by the mutator. Unfortunately, a read barrier is unnecessarily expensive; there is no need to protect the mutator from seeing an invalid version of a pointer. A read barrier also incurs a larger overhead, since heap reads are way more common than heap writes.

A *write barrier* is activated upon writes to heap objects. Catching heap writes is cheaper because they happen less frequently. Jones and Lins point out that over 90% of heap writes happen at allocation time; later writes into heap objects are quite rare [JL96].

Write-barrier approaches fall into two different categories, depending on which of the two conditions mentioned above they address:

Snapshot-at-beginning collectors ensure that the second condition cannot happen: the original reference cannot be lost.

Incremental-update collectors preserve the first condition: changes to the connectivity of the graph are caught.

I explain these techniques in more detail in the next two sections.

2.2.4 Snapshot-at-beginning

The first alternative to coordinating the mutator and the collector with a write barrier is the snapshot-at-beginning algorithm. Snapshot-at-beginning algorithms prevent the loss of the original reference to a white object by making a copy of the original reference. These algorithms do not preserve the coloring invariant. Instead, they guarantee that there is at least one path leading to each reachable white object.

When the mutator updates a pointer in a black object, the write barrier traps it and stores the original pointer for later inspection by the garbage collector. When the collector resumes, it colors the object grey to which the original pointer refers to. One of the best-known snapshot write-barrier algorithms is Yuasa's algorithm [Yua90]: When the mutator writes to a memory location, the write barrier first saves the overwritten value and pushes it on a marking stack for later examination. This guarantees that no objects will become unreachable to the garbage collector traversal—all objects live at the beginning of garbage collection will be reached, even if the pointers to them are overwritten.

Snapshot-at-beginning algorithms are very conservative: No objects that become garbage in one garbage collection cycle can be reclaimed in that cycle; all of the overwritten pointers are preserved and traversed. The garbage collector reclaims these objects at the next garbage collection cycle. Additionally, all newly allocated objects during a collection cycle are effectively allocated black even though the chance of a young object dying within a single collection cycle

may be high. Snapshot-at-beginning heaps cause significant memory overhead because they incur large amounts of floating garbage.

2.2.5 Incremental-update

Incremental-update methods are less conservative than snapshot algorithms. They incrementally record changes made by the mutator to the connectivity of the graph, rather than making a static estimate of the reachability at the start of a collection cycle. Incremental-update algorithms preserve the coloring invariant: they prevent Condition 1. The best known incremental-update algorithms are due to Dijkstra [DLM$^+$78] and Steele [Ste75]; they were developed independently and are quite similar.

When the client stores a pointer into an object that has already been passed by the traversal and thus colored black, it changes the graph of reachable data structures behind the collector's back. The incremental-update write barrier has to record such changes and notify the collector about black objects that currently hold pointers to white objects. The collector then reverts such objects to grey. This way, the traversal scans the formerly black objects again before garbage collection completes, to find any live objects that would otherwise escape.

In the example shown in figure 2.4, object A would be colored grey and traversed again when the garbage collection resumes. The traversal would then reach object C that would survive the sweep phase, resulting in a consistent heap without any dangling pointers.

Another flavor of the incremental-update mechanism exists that does not re-color the modified object grey, but colors the pointed-to white object grey to preserve the coloring invariant—in the example, object C would be colored grey, and A would remain black. This is a more conservative coloring strategy as it preserves C regardless of whether the pointer is subsequently deleted.

The performance overhead of an incremental traversal is as follows: Whenever a collection is resumed, the write-barrier information has to be read out. This means, the garbage collector has to re-color all the objects grey that are recorded by the write barrier. Also the collection of the write-barrier data adds extra cost to each heap write. The incremental-update write barrier increases the time spent in the mark phase, because already reached, modified objects have to be scanned again. On the other hand, the cost of the extra visits will reduce the amount of floating garbage left at the end of the collection cycle. Section 4.3 evaluates the measurements for performance and memory overhead.

The amount of floating garbage left behind by an incremental-update algorithm is also affected by its policy towards new cells. If the mortality rate of new cells is sufficiently high, many will die before they are reached by the traversal. New dead cells that were allocated white can be reclaimed in the same collection cycle, but cells allocated black or grey will survive the collection cycle, whether they are still visible to the mutator or not. The cost of allocating white or grey is that any newborn cells that do survive must be traversed.

2.3 Summary

XEmacs's annoying pause times are caused by the slow garbage collector that runs atomically while the client is stopped. Especially the traversal of the

reachability graph, which decides which objects are live and which are garbage, is taking up the most time of a collection cycle. To reduce the garbage collection pauses, an incremental scheme can help. Traditional mark-and-sweep garbage collection can be made incremental by interleaving collection with client activity. Care must be taken to ensure that the collector and the client are synchronized to make sure that no objects visible to the client are swept.

To interleave the client and the collector promises a more reactive system. However, there is a price to pay: the mutator and the collector have to coordinate. In particular, storing pointers into objects is more costly, and the modified objects have to be traversed repeatedly.

Chapter 3

Implementation

The first part of this chapter introduces different write-barrier approaches and describes implementations on different platforms. The write barriers make use of memory protection and signal handling. The second part of this chapter describes the changes made to the XEmacs source code to use the write barrier and the new incremental garbage collector.

Because of the discussed benefits in respect to directness, performance, and conservativeness, I use the incremental-update method that re-colors black objects upon write-barrier traps. Additionally, this technique interoperates better with the write-barrier implementation I use than the other algorithms described. I explain this decision in this chapter after the discussion of write-barrier implementations is complete.

3.1 Write Barrier

The previous chapter explained the need for coordinating the collector and the mutator with a write barrier to prevent the incremental traversal of the reachability graph from failure. This section gives an overview of different write-barrier implementations, and then describes implementations for different operating systems and machines in detail.

3.1.1 Overview of Mechanisms

A write barrier informs the collector about objects that are modified by the mutator while a incremental collection runs. There are several distinct approaches to implementing a write barrier [Boe]:

Explicit source code annotation: The most direct and most exact way to set up a write barrier is to annotate the source code explicitly: Every assignment in the sources has to be found and code has to be added to inform the collector. One way to achieve this would be to wrap a macro call around every pointer update, where the macro identifies the modified object and communicates its address to the garbage collector.

Unfortunately, this approach is not easily implementable in XEmacs: XEmacs contains several hundred thousand lines of code, which makes

it nearly impossible to identify all heap writes. Additionally, strict coding rules have to be adopted by all other XEmacs code contributors because all future written code also has to use the write barrier, which is not easily enforceable.

Another possibility would be the development of a tool that automatically annotates the code. This is outside the scope of my work, however.

Software write barriers impose an overhead on all pointer updates performed by the mutator. On many systems, the overhead in the mutator can be removed or at least reduced with assistance from the virtual memory. Virtual memory provides dirty-bit information for *logical memory pages* rather than for single objects. Modified objects and pages are called *dirty*—in an implementation this is often reflected by setting a *dirty bit* for the modified object. The following implementations make use of dirty-page information:

Virtual Dirty Bit: Most operating systems support a generic *virtual-dirty-bit mechanism*: The write barrier write-protects memory pages containing heap objects. If the mutator tries to modify these objects by writing into the write-protected page, the operating system generates a fault. The write barrier catches this fault, reads out the error-causing address and can thus identify the updated object and page. Not all environments provide the mechanism to write-protect memory, catch resulting write faults, and read out the faulting address [Boe].

Special operating-system support: Some operating systems, such as Sun's Solaris 2, provide dirty-bit information through the process file system `/proc`. The use of `/proc` involves reading the dirty bits from the entire address space. Filtering out the dirty bits of the current process may be slow [JL96].

External dirty bit information: Some external systems provide dirty-bit information that can be used by other applications. One example is the Xerox Portable Common Runtime that provides functionality for reading out dirty pages [Boe].

For maximum portability, I chose the virtual-dirty-bit write-barrier implementation.

The virtual-dirty-bit write barrier does depend on operating-system-specific functionality, but the basic strategy is the same on all systems. Fortunately, nearly all of today's operating systems provide the needed features. Albeit, there may be big differences how the virtual-dirty-bit implementation works on different systems. The next sections provide the details and the prerequisites of the virtual-dirty-bit write barrier and describe how the write-barrier implementation differs among most common operating systems.

3.1.2 Memory Protection

Memory protection is the first prerequisite of a virtual-dirty-bit write barrier: The system has to provide a method for setting and removing write-protection for certain memory blocks. Such a feature is provided by most current operating

systems like UNIX/Linux, Windows, and Mac OS. It is part of the POSIX standard (IEEE 1003.1c) [IG04].

Memory protection does not work for arbitrary small memory regions; it only works in terms of *pages*. Memory provided by the operating system is broken into blocks of the same size called pages. The *page size* is usually defined by the hardware, the size of a page is a power of 2, varying between 4,096 bytes and 4,194,304 bytes, depending on the computer architecture and machine model [SGG03]. The `PAGESIZE` system variable that the operating system defines specifies which page size in bytes the system uses. Implementations of a memory-protection mechanism may restrict the size and the alignment of the memory region to be on page-size boundaries. If an implementation has no restrictions on size or alignment, it may specify a one-byte page size [IG04].

All objects subject to be covered by the write barrier have to be allocated on logical memory pages, so that they meet the requirement to be write-protected. In XEmacs, all objects of the Lisp engine, called *Lisp objects*, have to be allocated on such pages. The allocator [Cre04] is aware of a system page size—it allocates all Lisp objects on logical memory pages.

Since memory pages are large compared to most Lisp objects, many objects may reside on a single memory page, which leads to coarse granularity of the memory-protection mechanism. This causes an overhead that is described in the next section.

Once a memory area is write-protected, write access to it causes a fault that is signaled to the process. Every time the mutator updates a write-protected Lisp object, the write access provokes a fault. Handling these faults generated by illegal memory access is another requirement of a virtual-dirty-bit write barrier. It is discussed in the next section.

3.1.3 Signal Handling

This section addresses the second prerequisite of a virtual-dirty-bit write barrier: signal handling. A *signal* is used to notify a process that a particular event has occurred [SGG03]. Examples of such events include detection of hardware faults, timer expiration, division by zero, and illegal memory access. All signals follow the following pattern:

1. A signal is generated by the occurrence of a particular event.
2. A generated signal is delivered to a process.
3. Once delivered, the signal must be handled.

Every signal has a default function that handles the signal, called *signal handler*. This default action may be overridden by a user-defined signal-handler function. In this instance, the user-defined function is called to handle the signal rather than the default action.

For the implementation of a virtual-dirty-bit system, the write barrier has to install its own signal handler function to catch the signals caused by illegal memory access. Usually, the default action on illegal memory access is to kill the process that caused the fault and print an error message. The signal handler of the write barrier instead has to identify the Lisp object that caused the illegal

memory access, inform the garbage collector about the updated object, and ensure that the desired change can be written to memory.

When a signal handler completes without killing the process, control is returned to the client where it was interrupted. In our case the client was interrupted during a heap write, typically an assignment, which has not completed yet. To have it complete successfully, the signal handler has to remove the write-protection from the memory area the Lisp object resides in before the client gets control back.

As stated in the previous section, memory-protection operations only work on logical pages. Therefore the write-protection is removed from the page as a whole rather than the single modified Lisp object. Since the granularity of the memory protection is coarse, many Lisp objects may be affected by removing the protection. This leads to an unavoidable hole in the write barrier: It cannot catch further writes to the Lisp objects on this page. Consequently, the garbage collector has to treat all objects on this page as modified. This causes overhead: The garbage collector has to re-scan objects that have not been modified by the client, merely because they reside on the same page as a modified object.

Removing the write-protection from a page also removes the overhead caused by the write-barrier signal handling for future writes. My experiments show that a write-barrier heap write takes approximately twice as long as a heap write without the write barrier:

- 100,000 heap writes with write barrier take 952 milliseconds, and
- 100,000 heap writes without write barrier take 511 milliseconds.

Calling the signal handler and removing the write protection takes up significantly more time than heap writes without protection. But the overhead for a heap write with running write barrier is swapped for the overhead of re-examining unchanged objects: With this taken into account, the resulting overhead is not too big—the incremental garbage collector is actually only about 12% slower than the traditional one. More performance evaluations can be found in section 4.3.3.

3.1.4 Faulting Address

To identify the modified object and page, the write barrier needs to be able to retrieve the faulting address of an illegal memory access. This is the third requirement of a virtual-dirty-bit write barrier. Unfortunately, many systems do not provide this feature.

The POSIX standard [IG04] describes a signal information data structure that is passed to the fault handler. The data structure contains a field with the fault-causing address. Some operating systems do not provide this field or do not provide the data structure at all, even though they are purportedly POSIX-compliant.

Some systems may provide alternative ways to retrieve the faulting address, such as values retrieved directly from the kernel, or reading out registers of the processor.

On systems that completely fail to provide the faulting address the virtual-dirty-bit write barrier and the incremental garbage collector do not work.

Once the write barrier retrieved the faulting address, it needs to identify the modified object and the modified page. This mapping is done by a two-level search tree implemented in the allocator [Cre04, RRSF00]: The upper and lower bits of the faulting address are used to index into the two levels of the search tree, which in turn leads to the address of the page header and the page itself. With the information found in the page header the write barrier identifies the object. The write barrier then marks the object and page dirty, and informs the garbage collector. This is described in detail in section 3.3.

3.1.5 Choosing a Write-Barrier Algorithm

A disadvantage of the virtual-dirty-bit technique is the coarse granularity of the memory-protection mechanisms. This directly influences the choice of a write-barrier algorithm as it leads to an overhead due to re-examining unmodified objects, as all objects of a page are subject to re-examination by the traversal when the protection of the page is removed.

The snapshot-at-beginning approach—described in section 2.2.4—makes a copy of the overwritten pointers. With the virtual-dirty-bit implementation, the approach would have to make copies from all the pointers on a page. This is not feasible: First, the identification of the pointers of all objects is costly; second, large amounts of memory may have to be used to store the copies. Hence, snapshot-at-beginning does not work well with the virtual-dirty-bit technique.

The incremental-update method—see section 2.2.5—re-colors modified black objects grey. With the virtual-dirty-bit implementation, all black objects on a page have to be changed to grey. Coloring grey usually corresponds to pushing the object's address on the mark stack in a real-life implementation. This is cheap.

3.2 Platform-Dependent Implementation

Most of today's operating systems provide the features needed for the previously described write-barrier implementation. However, they provide them in different ways. This section provides the details how the write-barrier implementation is accomplished on different platforms.

My implementations are based on ideas and hints from implementations of PLT's MzScheme [PLT] and the Boehm-Demers-Weiser conservative garbage collector for C and C++ [Boe]. Information about the system calls on UNIX/Linux platforms is taken from the system manual pages [man], the POSIX standard [IG04], and C manuals [KR88, FSF01].

3.2.1 POSIX-Compliant Platforms

POSIX is an acronym for *Portable Operating System Interface* [Wik05]. It describes a standard set of behaviors and system calls for UNIX-like operating systems. POSIX is the collective name for a set of standards specified by the *Institute of Electrical and Electronics Engineers*, short IEEE, for UNIX-like systems. The standards are formally designated as IEEE Std 1003. The POSIX standard was developed, and is maintained, by a joint working group of members

of the IEEE Portable Applications Standards Committee, members of The Open Group, and members of ISO/IEC Joint Technical Committee [IG04].

POSIX provides a common baseline for UNIX-like systems. On POSIX-compliant platforms, there is a straightforward way to implement the virtual-dirty-bit write barrier:

Memory protection: POSIX defines the `mprotect(2)` system call to control access to a region of memory:

```
int mprotect (const void *addr, size_t len, int prot);
```

The `mprotect` function specifies the desired protection for the memory area of the interval [`addr`, `addr` + `len`[. The value of `addr` has to be on `PAGESIZE` boundaries, and `len` has to be a integer multiple of `PAGESIZE`.

The protection `prot` is a bitwise-or of the following values:

`PROT_NONE`	The memory cannot be accessed at all.
`PROT_READ`	The memory can be read.
`PROT_WRITE`	The memory can be written to.
`PROT_EXEC`	The memory can contain executing code.

The parameter `prot` is either `PROT_READ` to remove the write access by setting the memory read-only, or it is set to `(PROT_READ | PROT_WRITE)` to allow read and write access. If an access is disallowed by the given protection, the program receives a signal.

Signal handling: The signal that the kernel generates when an illegal memory access occurs is one of the following two [IG04]:

`SIGSEGV` for "segmentation violation," better known as "segmentation fault," or

`SIGBUS` for "bus error"

The signal handler has to be installed on both signals. To install a signal handler according to POSIX, the write barrier has to use the `sigaction(2)` system call:

```
int sigaction (int signum, const struct sigaction *act,
                           struct sigaction *oldact);
```

The `signum` parameter specifies the signal the new signal action `act` is installed for. The previous action is saved in `oldact`.

The signal action structure `struct sigaction` contains a field `sa_sigaction` that specifies the action to be associated with `signum`. It contains a pointer to a fault-handler function with the following signature:

```
void fault_handler (int, struct siginfo *, void *);
```

When the fault handler is invoked, it receives the signal number as its first argument, a pointer to a `struct siginfo` as its second argument and a pointer to a `ucontext_t` (cast to `void *`) as its third argument. Here, the `ucontext_t` is ignored; `struct siginfo` is more important, it contains the faulting address.

Faulting address: The fault handler's second argument is a data structure called `struct siginfo`. It contains a field `si_addr`, which is set to the address that caused the fault. The faulting address can be retrieved by accessing `si_addr`.

While the memory-protection and signal-handling interfaces described here can be used on all POSIX-based platforms, the POSIX way to retrieve the faulting address is not implemented on all platforms that call themselves POSIX-compliant—see the implementation details for Mac OS X in section 3.2.3 and Cygwin in section 3.2.5.

3.2.2 Old UNIX/Linux

Older UNIX-like systems and older Linux systems[1] do not provide the POSIX signal handling interface. These systems provide a restricted version: the `signal(2)` system call.

Memory protection: The write barrier uses the `mprotect(2)` system call to set the protection status, as described in section 3.2.1.

Signal handling: The `signal(2)` system call installs a new signal handler:

```
sighandler signal (int signum, sighandler handler);
```

The system call installs the new `handler` on the signal `signum`. The signals the write barrier has to catch are `SIGSEGV` and `SIGBUS`. The `handler` is of type `sighandler`, which is a pointer to function defined as follows:

```
typedef void (*sighandler) (int, struct sigcontext);
```

A signal handler is a function, that takes two arguments: the signal number and a `struct sigcontext` that contains additional information about the signal.

The `signal` function returns the previously installed signal handler.

Faulting address: The fault handler's second argument is a data structure called `struct sigcontext`. It contains a field `cr2`, which is set to the address that caused the fault, when the signal handler is invoked. The faulting address can be retrieved by accessing `cr2`.

On modern UNIX/Linux systems this simple `signal(2)` mechanism is also available, but where possible one should use `sigaction(2)` instead [IG04].

3.2.3 Mac OS X

Mac OS X is based on a BSD Mach microkernel [App]. The Mach kernel does not provide a POSIX-compatible method to obtain the faulting address. Instead, it provides a more powerful fault handling facility, called the Mach Exception Handler. On Mach systems, the POSIX signal handling gets emulated [SGG04].

[1] Linux kernel version < 2.4

The kernel translates all Mach exceptions to POSIX signals, but the field `si_addr` of the `siginfo` data structure is not set [Boe].

To compensate this, there is a workaround that allows a POSIX fault handler to read out the `dar`-field of the exception state—but this workaround fails when running in the debugger. Thus, for the implementation of the write barrier I chose to drop down to the Mach level. In the following I provide a rough outline how the write barrier makes use of the Mach Exception Handler. For a thorough treatment of the Mach Exception Handlers, see the work of Black and others [BGH+88].

Memory protection: The write barrier is still able to use the `mprotect(2)` system call to set the memory-protection status, as described in section 3.2.1. Now the resulting faults are not caught with POSIX signals, but with Mach exceptions.

Signal handling: To use the Mach exceptions, the write barrier has to start its own exception-handling thread and set up the communication with this thread using communication channels called exception ports. The exception thread has to read the exception from the Mach exception server and forward the `KERN_PROTECTION_FAILURE` exception—this is the Mach equivalent for illegal memory access—to the write-barrier exception handler.

Faulting address: The exception handler has access to a data structure called `exception_data`. The faulting address can be retrieved from `exception_data`.

3.2.4 Native Windows

To compile XEmacs natively for Windows, a Microsoft C/C++ compiler is required. On native Windows, the virtual-dirty-bit write barrier has to use the Microsoft Exception Handling Mechanisms. Microsoft calls its *Structured Exception Handling* an "extension to Microsoft C/C++" [Pre00b].

Memory protection: The `VirtualProtect` Windows system call changes the protection on a region of pages in the virtual address space of the calling process [Pre00c].

Signal handling: To install an exception handler, I use `AddVectoredExceptionHandler`. It adds the given fault handler—`VectoredHandler` in Windows terms—to the front of a list of exception handlers. When an exception occurs, the handlers in this list are called in order. If the write-barrier handler is called, it checks if the exception is an `EXCEPTION_ACCESS_VIOLATION`, which is caused by illegal memory access. In this case, the handler is really handling this exception and returns `EXCEPTION_CONTINUE_EXECUTION` to tell the Windows exception mechanism that the exception is taken care of and program execution can be continued; no further handler has to be called. Otherwise the handler returns `EXCEPTION_CONTINUE_SEARCH` to indicate that an other handler has to take care of the current exception [Pre00a].

Faulting address: The exception handler has access to a data structure called `ExceptionRecord`. The faulting address can be retrieved from the field `ExceptionInformation` of this structure.

3.2.5 Cygwin

Cygwin is a Linux-like environment for Windows—it provides a POSIX-compatible interface [Cyg]. It aids porting software that runs on POSIX systems to Windows. The Cygwin version of XEmacs is an alternative to the native Windows build.

Unfortunately, Cygwin does not implement all of POSIX yet; specifically, it lacks a way to read out the faulting address. The field `si_addr` of the `siginfo` data structure passed to the signal handler is containing bogus values. This is a known bug—the Cygwin maintainers are working on it [Cyg].

Luckily, Cygwin also provides access to the underlying native Windows application programming interface. Consequently, my implementation of the write barrier under Cygwin uses the mechanisms described for native Windows in section 3.2.4. Unfortunately, this approach has a downside: Using a debugger on a Cygwin-built XEmacs with the virtual-dirty-bit write barrier does not work: Cygwin-based debuggers, such as Cygwin's `gdb`, conflict with the native Windows exception handling; and Windows-based debuggers, such as the Microsoft Visual Studio Debugger, do not produce usable results with Cygwin-compiled executables.

The current situation is a temporary workaround and will be fixed as soon as Cygwin's implementation of the `siginfo` structure is complete.

3.2.6 Debugging the Write Barrier

The virtual-dirty-bit write barrier provokes signals on purpose, namely `SIGSEGV` and `SIGBUS`. When debugging XEmacs with this write barrier running, the debugger always breaks whenever a signal occurs. This behavior is generally desired: A debugger has to break on signals, to allow the user to examine the cause of the signal—especially for illegal memory access, which is a common programming error.

But the debugger should not break for signals caused by the write barrier. Therefore, most debuggers provide the ability to turn of their fault handling for specific signals. The following command prevents `gdb` from stopping and printing information when `SIGSEGV` or `SIGBUS` occurs:

```
handle SIGSEGV SIGBUS nostop noprint
```

I added this command to `gdb`'s initialization file `.gdbinit` in the XEmacs sources directory: whenever `gdb` is started to debug XEmacs with the virtual-dirty-bit write barrier enabled, it does not break on signals caused by illegal memory access.[2]

But what happens if a bug in XEmacs causes an illegal memory access? To maintain basic debugging abilities, I use another signal: First, the write-barrier signal handler has to determine if the current error situation is caused by the

[2] For the `dbx` debugger, I added '`ignore SIGSEGV SIGBUS`' to `.dbxrc`.

write-barrier memory protection or not. Therefore, the signal handler checks if the faulting address has been write-protected before. If it has not, the fault is caused by a bug; the debugger has to break in this situation. To achieve this, the signal handler raises `SIGABRT` to abort the program. Since `SIGABRT` is not masked out by the debugger, XEmacs aborts and allows the user to examine the problem.

Using a debugger on Windows does not need any precautions: Under Windows, as seen in section 3.2.4, the debugger's exception handler is not called when the write barrier's exception handler returns `EXCEPTION_CONTINUE_EXECUTION`, because program execution resumes immediately. If the exception was not caused by the write barrier, returning `EXCEPTION_CONTINUE_SEARCH` will eventually call the debugger's handler, which then breaks.

Using `gdb` on Mac OS X does also not need any precautions—the Mach exception handlers occur on a lower level and never reach the debugger. Only if the exception is not handled by the write barrier—if the illegal memory access is not caused by the virtual-dirty-bit memory protection—the system translates the exception to `EXC_BAD_ACCESS`, which the debugger catches.

Debugging on Cygwin is currently not possible, see section 3.2.5.

3.3 Changes to XEmacs

In this section I describe the changes I made to the XEmacs source code.

First, configuration and startup issues are resolved. Then, the steps of garbage collection are illustrated and implementation details for each step are provided.

3.3.1 Configuration

The new incremental garbage collector is fully conditioned on the preprocessor symbol `NEW_GC`. To enable the incremental collector, the argument `--enable-newgc` has to be given to the `configure` script on systems that support the `autoconf`/`automake`-environment:

```
./configure --enable-newgc
```

This implicitly enables the prerequisites of the new garbage collector:

- the new allocator `MC_ALLOC`, and
- the new mark algorithm `USE_KKCC`.

The configuration script determines which virtual-dirty-bit write-barrier implementation to use: It checks if the POSIX interface `sigaction(2)` and `struct siginfo` with the field `si_addr` is available, or if the `signal(2)` and `struct sigcontext` with field `cr2` interface has to be used.

The configure script recognizes Mac OS X and Cygwin and uses the special case write-barrier implementations.

On systems that are not explicitly listed in the `configure` script, it always checks for `sigaction(2)` and `signal(2)` availability. Consequently, the write barrier runs out-of-the-box on all fully POSIX-compliant systems.

Native Windows has a special configuration file `nt/config.inc`. To enable the new garbage collector on native Windows, the configuration file must contain the following lines:

```
USE_KKCC=1
MC_ALLOC=1
NEW_GC=1
```

If a systems fails to provide the functionality needed by the write barrier, a fall-back "fake" implementation is used: This implementation simply turns of the incremental write barrier at runtime and does not allow any incremental collection. The garbage collector then acts like a traditional mark-and-sweep garbage collector. Generally, the incremental garbage collector can be turned off at runtime by the user or by applications. This feature is described in more detail in section 3.3.19.

3.3.2 Startup

The installation of the write-barrier signal handler is among the first things XEmacs does when it starts. XEmacs installs own signal handlers on all signals that cause program abortion, such as illegal memory access. These signal handlers are used to print a Emacs Lisp backtrace that provides important information to identify crashes caused by Lisp code. To keep this functionality, the write-barrier signal handler is installed after the backtrace signal handlers. It stores the previously installed backtrace handler and resurrects it, when the write-barrier signal handler is invoked by a invalid memory access that is not caused by the write barrier. This way, basic debugging information in case of an signal-caused program abortion remains available. The signal handler of the write barrier is described in more detail in section 3.3.11.

3.3.3 New Lisp Objects

Some Lisp objects do not carry all their information in the object itself. External parts are kept in separately allocated memory blocks that are not managed by the allocator. Examples for these objects are hash tables and dynamic arrays, two objects that can dynamically grow and shrink.

The separate memory blocks are not guaranteed to reside on page boundaries, and thus cannot be watched by the write barrier. Moreover, the separate parts can contain live pointers to other Lisp objects. These pointers are not covered by the write barrier and modifications by the client during garbage collection do escape. In this case, the client changes the connectivity of the reachability graph behind the collector's back, which eventually leads to erroneous collection of live objects.

To solve this problem, I transformed the separately allocated parts to fully-qualified Lisp objects that are managed by the allocator and thus are covered by the write barrier. This also removes a lot of special allocation and removal code for the out-sourced parts. Generally, allocating all data structures that contain pointers to Lisp objects on one heap makes the whole memory layout more consistent.

In the following, I demonstrate the needed steps to transform an external data structure to a Lisp object, as described in the XEmacs Internals Manual

[WTB+], considering the new Lisp object `Hash_Table_Entry` as an example that is described later in this section:

- Add the field `lrecord_header` as the first entry to the object's declaration:

```
typedef struct htentry
{
  struct lrecord_header lheader;
  Lisp_Object key;
  Lisp_Object value;
} htentry;
```

  A hash table entry consists of a key and a value, and the newly added Lisp object header.

- Add macros to declare and access the Lisp object and the object's fields:

```
DECLARE_LRECORD (hash_table, Lisp_Hash_Table);

#define XHASH_TABLE(x) \
  XRECORD (x, hash_table, Lisp_Hash_Table)
#define wrap_hash_table(p) wrap_record (p, hash_table)
#define HASH_TABLEP(x) RECORDP (x, hash_table)
#define CHECK_HASH_TABLE(x) CHECK_RECORD (x, hash_table)
#define CONCHECK_HASH_TABLE(x) \
  CONCHECK_RECORD (x, hash_table)
```

- Create a memory description for the new object. Memory descriptions are used to mark the object, see section 3.3.7.

```
static const
struct memory_description htentry_description_1[] = {
  { XD_LISP_OBJECT, offsetof (htentry, key) },
  { XD_LISP_OBJECT, offsetof (htentry, value) },
  { XD_END }
};
```

- Define the implementation structure of the object that describes the object's properties using `DEFINE_LRECORD_IMPLEMENTATION`.

```
DEFINE_LRECORD_IMPLEMENTATION ("hash-table-entry",
                               hash_table_entry,
                               1, 0, 0, 0, 0, 0,
                               htentry_description_1,
                               Lisp_Hash_Table_Entry);
```

- To initialize the new implementation, I added

```
INIT_LRECORD_IMPLEMENTATION (hash_table_entry);
```

  to the hash table's initialization function `init_elhash_once_early`.

- Add a unique type identifier to the enumeration `enum lrecord_type`, in this case `lrecord_type_hash_table_entry`.

In the following I provide more details to some of the external data structures that I have transformed to Lisp objects:

Dynamic Arrays: A `dynarr` is a contiguous array of fixed-size elements with no upper limit on the number of elements in the array—except available memory. Because the elements are contiguous, random access to a particular element is constant-time. In the old garbage collector, dynamic arrays held non-Lisp objects as well as Lisp objects. The new incremental garbage collector distinguishes between dynamic arrays that hold Lisp objects and dynamic arrays that hold non-Lisp objects.

For dynamic arrays that hold Lisp objects, I created the new Lisp object type "Dynamic Lisp Array," called `dynarr_lisp`. It contains Lisp objects, in contrast to the dynamic array `dynarr` that contains non-Lisp objects. A `dynarr_lisp` can be created with the new macro

```
Dynarr_lisp_new (type, dynarr_imp, imp);
```

where `type` and `dynarr_imp` are previously defined dynamic-array-type and implementation structures, and `imp` is the Lisp object implementation of the objects stored in this instance of the array.

The rest of the interface remains the same: adding and removing elements from dynamic Lisp arrays does not differ from how it is described for dynamic arrays in the XEmacs Internals Manual [WTB+].

For dynamic Lisp arrays, the allocator has to be extended: I added two more primitives to the new allocator to allow allocation of elements in a contiguous block of memory: `mc_alloc_array` and `alloc_lrecord_array`. Each element in the array is a regular Lisp object with its own Lisp object header and mark bit. The internal allocation function

```
void *mc_alloc_array (size_t size, int elemcount);
```

allocates `elemcount` elements of given `size` in one contiguous block of memory. It returns a pointer to the block.

The function

```
void *alloc_lrecord_array (Bytecount size,
                           int elemcount,
                           const struct
                           lrecord_implementation *imp);
```

calls `mc_alloc_array` with `size` and `elemcount`, and initializes every object in this array with the lrecord implementation `imp`. This way the entries of the array become regular Lisp objects. See the interface descriptions in appendix A.

Newly created Lisp types for dynamic Lisp arrays are `Lisp_Merged_Faces`, `Lisp_Face_Cachels`, `Lisp_Glyph_Cachels`, and `Lisp_Display_Blocks`.

Hash Tables: Hash tables are internally represented in two parts: header information is stored in a Lisp object, and the actual data is allocated in a separate memory block. With the old garbage collector, the separate memory block was not allocated on the Lisp heap. But hash table entries contain Lisp objects, therefore the hash tables with all entries have to be allocated on the Lisp heap to have them covered by the write barrier.

I added the new Lisp object `Hash_Table_Entry`. Objects of this type are allocated with the new array allocation primitives `mc_alloc_array` and `alloc_lrecord_array`, described above and in appendix A.

The hash tables remain split up in two parts, but now both parts are regular Lisp objects.

Consoles, Devices, Frames, Windows: A window that the user sees on the screen is called *frame* in XEmacs terminology. Each frame is subdivided into one or more non-overlapping panes, called *windows*. Frames and windows are displayed on *devices*. *Consoles* in XEmacs terminology represent instances of keyboards, like a keyboard of the local X Server or from a remote TTY [WTB+].

All these objects carry a `data` field that contains information for a specific instance of the current object. This data is allocated separately and contains live pointers to objects on the Lisp heap. The instances of these objects are, for example, `x` for the X Server, `tty` for terminals, `msw` for Windows instances, and `stream` for stream-oriented input/output. I transformed all instances of all objects to Lisp objects, so that the write barrier can cover them.

I transformed all data structures that contain live pointers to Lisp objects and moved them to the Lisp heap. Consequently, the write barrier covers all cells containing live pointers. The write barrier does not have any holes left, no modification of a pointer can escape.

This is a prerequisite for the incremental garbage collector to work. The new garbage collector is described step-by-step in the next section.

3.3.4 Garbage Collection—step-by-step

Figure 3.1 illustrates the incremental garbage collection scheme used by the implementation. Each operational step of the new incremental garbage collector is described in its own section:

1. invocation of garbage collection in section 3.3.5,
2. preparation in section 3.3.6,
3. mark phase in section 3.3.7,
4. mark root set in section 3.3.8,
5. traverse live objects in section 3.3.9,
6. interrupt mark phase in section 3.3.10,
7. write barrier in section 3.3.11,

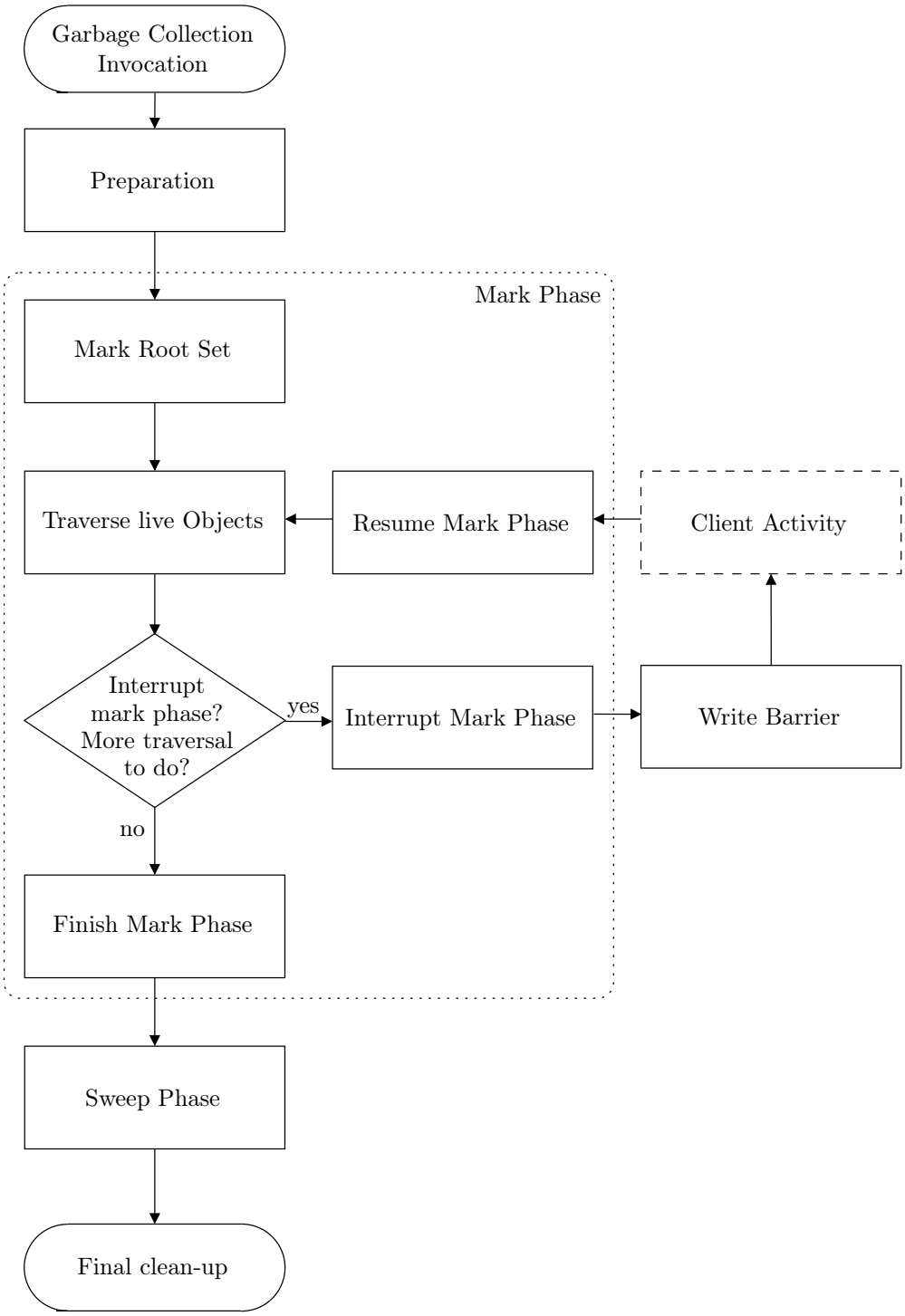

Figure 3.1: Implementation of incremental garbage collection

8. resume mark phase in section 3.3.12,
9. finish mark phase in section 3.3.13,
10. sweep phase in section 3.3.14, and
11. final clean up in section 3.3.15.

The garbage collection control is described in section 3.3.16.

3.3.5 Invocation of Garbage Collection

The entry point of every garbage collection is the `gc` function on C level. Two types of garbage collection exist:

Full collection: A full garbage collection runs atomically without any interruption. It guarantees that unused objects are freed.

Cycle of incremental collection: The collector makes incremental traversal work. One incremental run of the garbage collector is called cycle. After the cycle completes, the collector is suspended and the client resumes. One cycle does not necessarily free any memory, it only guarantees that the traversal of the heap makes progress.

The invocation can be triggered explicitly by calling the Lisp functions `(gc-full)` for a full collection or `(gc-incremental)` for a cycle of incremental garbage collection or it can occur implicitly in four different situations:

1. In the `main` function of XEmacs that is called at each startup, the garbage collection is invoked after all initial creations are completed, but only if the symbol `ERROR_CHECK_GC` for internal error checking is defined.
2. The `disksave_object_finalization` function of the portable dumper clears the objects from memory which need not be stored with XEmacs when image is dumped out.
3. Each time the function `eval` is called to evaluate a Lisp form, a garbage collection could happen.

   Three global variables control when garbage collection occurs:

   `consing_since_gc` counts the allocated bytes since the last garbage collection.

   `gc_cons_threshold` is a threshold given in bytes that triggers a new garbage collection when `consing_since_gc` exceeds `gc_cons_threshold`.

   `gc_cons_incremental_threshold` is a specified threshold given in bytes that triggers the next cycle of a running garbage collection when `consing_since_gc` exceeds `gc_cons_incremental_threshold`.

   After object allocation, the function `recompute_need_to_garbage_collect` is called that sets another global variable `need_to_garbage_collect` to true if the above thresholds are exceeded:

```
need_to_garbage_collect = write_barrier_enabled ?
  (consing_since_gc > gc_cons_incremental_threshold) :
  (consing_since_gc > gc_cons_threshold);
```

If an incremental garbage collection runs, the variable `write_barrier_enabled` is true and `consing_since_gc` is compared to `gc_cons_incremental_threshold`. If the write barrier does not run, `consing_since_gc` is compared to `gc_cons_threshold` that then determines the need to collect. If the thresholds are exceeded, `need_to_garbage_collect` is set to true.

If `need_to_garbage_collect` is true when `eval` is called, it invokes the next cycle of an incremental garbage collection or starts a new incremental garbage collection if no collection is currently running.

4. Each time the function `funcall` is called to process a Lisp function call, a garbage collection could happen. The invocation of a garbage collection works according to `eval`: If `need_to_garbage_collect` is true when `funcall` is called, it invokes the next cycle of an incremental garbage collection or starts a new incremental garbage collection if no collection is currently running.

Garbage collection can occur upon calls to `eval` or `funcall` once a certain amount of memory has been allocated since the last garbage collection. As calls to these functions are hidden in various other functions and the Lisp engine uses these functions to interpret the Lisp code, garbage collection can occur nearly everywhere and at any time.

The frequency of garbage collection can be influenced by modifying the thresholds. See section 3.3.19 for more details.

In the following sections, I describe exactly what happens upon garbage collection.

3.3.6 Preparation

Some preparations are needed to start a garbage collection:

1. There are several cases in which a garbage collection is not allowed to run, and the garbage collector is left immediately: when `gc_in_progress` indicates that a collection already runs, when `gc_currently_forbidden` indicates that garbage collection is somehow forbidden, for example when the display is currently updated, or when XEmacs is shutting down due to an unexpected failure—XEmacs "prepares for Armageddon" when the variable `prepare_for_armageddon` is true.

2. Determine the current frame in which all output that may occur during a garbage collection is put, and the current state of this frame is saved.

3. Before the traversal actually starts, references to objects that are no longer used are pruned: events, specifiers and the buffer undo list.

4. The variable `gc_in_progress` is set to indicate that a collection is in progress.

The next step is to start the mark phase.

3.3.7 Mark Phase

The incremental garbage collector is based on a mark implementation known as *KKCC* [Cre04]. The mark algorithm uses an explicit stack that keeps track of the current progress of the traversal. Objects that are pushed on the mark stack are the grey objects of an incremental traversal scheme.

The mark stack is accessed with the following functions:

```
void kkcc_gc_stack_push (void *ptr,
                         const struct memory description *desc);

kkcc_gc_stack_entry *kkcc_gc_stack_pop (void);
```

The first function pushes the Lisp object pointed to by `ptr` and its memory layout description `desc` onto the mark stack. The second one pops a `kkcc_gc_stack_entry` from the mark stack. A `kkcc_gc_stack_entry` consists of two fields: `ptr` contains the address of the object and `desc` contains the object's memory layout description.

Every Lisp object type has a memory layout description. Some examples:

The most basic Lisp object is the `cons` cell. A `cons` cell contains two Lisp objects, known as `car` and `cdr`. Its definition is:

```
struct Lisp_Cons
{
  struct lrecord_header header;
  Lisp_Object car;
  Lisp_Object cdr;
};
```

The `header` field contains type information, the `car` and `cdr` fields contain references to other Lisp objects, in other words: live pointers. The memory description holds the layout information of a `cons`:

```
static const struct memory_description cons_description[] = {
  { XD_LISP_OBJECT, offsetof (Lisp_Cons, car) },
  { XD_LISP_OBJECT, offsetof (Lisp_Cons, cdr) },
  { XD_END }
};
```

The memory description lists the two fields `car` and `cdr` and provides information about their contents: Both contain a reference to a Lisp object, indicated by the constant `XD_LISP_OBJECT`. The exact position of the fields is calculated by the `offsetof` function. This way, the mark algorithm is able to identify and examine the elements. The KKCC mark algorithm parses the memory description to determine all live pointers.

Objects processed by the incremental traversal are marked white, grey, and black by the mark algorithm. Therefore, the allocator provides mark bits for every object. These mark bits are kept in the page header of each page. The allocator provides functionality to set and read the mark bits: `MARK_WHITE (ptr)`, `MARK_GREY (ptr)`, and `MARK_BLACK (ptr)` to set the mark bit for the object pointed to by `ptr`; and `MARKED_WHITE_P (ptr)`, `MARKED_GREY_P (ptr)`,

and `MARKED_BLACK_P (ptr)` that evaluate to true if the mark bit is set to the appropriate color.

The mark phase begins with examination of the root set.

3.3.8 Mark Root Set

Next the mark phase marks all accessible elements. All elements of the roots of accessibility are initially pushed onto the mark stack. There are several distinct mechanisms XEmacs uses to keep track of the root set. The roots are traversed in this order:

- All constants and static variables containing Lisp objects that are registered via the `staticpro` function.
- All other Lisp objects that are registered via the `mcpro` function. Objects that have formerly been read-only did not have to be added to the root set, because the read-only property prevented them from being freed. The new allocator does no longer use the read-only property, so these objects have to be added to the root set with `mcpro` [Cre04].
- All local variables containing Lisp objects that are created in C functions and are registered via `GCPRO`.
- All local variables that are bound during the evaluation by the Lisp engine are pushed on the `specbind` stack.
- All catch blocks that the Lisp engine encounters during the evaluation are stored in the `catchlist`.
- Every function application is pushed onto the backtrace stack of the Lisp engine. The backtrace stack serves as a root of accessibility.
- All objects created for profiling purposes are allocated by C functions. These objects are also roots.

The root set is pushed onto the mark stack in one atomic action that is not interrupted.

As the root set does not reside in the heap, the write barrier cannot cover it. Thus, there is no way to catch modifications to the root set when the client runs during a garbage collection. The only solution is to re-examine the the root set again at the very end of the mark phase. This is described in section 3.3.13.

3.3.9 Traverse Live Objects

The traversal of live objects works according to the algorithm described in section 2.2.1:

All objects are initially colored white. Objects that are reached by the traversal are pushed onto the mark stack and colored grey. The roots of accessibility are the first objects that have been pushed onto the mark stack. Next, the mark function

```
void gc_mark (int incremental);
```

is called. If `incremental` is > 0, `incremental` steps of traversal work are done, before the garbage collector rests. If `incremental` is ≤ 0 the traversal is executed in one atomic action without any interruption. If the collector works in incremental mode, the standard value of `incremental` is set to the value of the global variable `gc_incremental_traversal_threshold`. The variable contains how many steps of incremental work have to be executed in one incremental traversal cycle. One step of traversal work consists of the following actions:

1. Pop Lisp object `obj` and its description `desc` from the mark stack.

2. If `obj` is already marked grey or black, nothing has to be done: Skip actions 3–5.

3. Otherwise, color `obj` black.

4. Parse the memory description `desc` and find each live pointer `obj` contains.

5. Push each live pointer inside the object onto the mark stack and color the object pointed to grey.

These actions iterate until the mark stack is empty or `incremental` steps have been made. When this threshold is reached before the stack is empty, the mark phase is interrupted and the client is resumed, see the next section 3.3.10. If the mark stack is empty, the mark phase is finished and the rest of the garbage collection is executed without interruption, as described in section 3.3.13.

3.3.10 Interrupt Mark Phase

When `gc_mark` returns with an non-empty mark stack, the current garbage collection is interrupted to give the control back to the client. Before the client can resume, the write barrier needs to be set up: the heap has to be write-protected. The function

```
void protect_heap_pages (void);
```

scans the entire heap and write-protects every page that contains black objects. Only black objects have to be protected by the write barrier, because the collector assumes that it is finished with black objects. Grey objects are currently on the mark stack and do not require a write barrier. White objects have not yet been reached by the traversal, so they also do not have to be covered by the write barrier.

Consequently, only pages with black objects are write-protected—of course, these pages may also contain grey or white objects—but these objects are not re-pushed on the mark stack when the client is resumed in case this page gets modified by the client.

Once the heap is write-protected, the write barrier is activated—any write access invokes the write-barrier fault handler. Next the variable `write_barrier_enabled` is set to true.

3.3.11 Write Barrier

To understand how the write barrier works, it is best to take a look at the fault handler. The following listing shows the POSIX fault handler:

```
void
vdb_fault_handler (int signum, struct siginfo *siginfo,
                   void *UNUSED (ctx))
{
  if (write_barrier_enabled
      && (fault_on_protected_page (siginfo->si_addr)))
    {
      vdb_designate_modified (siginfo->si_addr);
      unprotect_page_and_mark_dirty (siginfo->si_addr);
    }
```

First, the fault handler has to check if the illegal memory access is really caused by the write barrier or if it is caused by something else—most likely an XEmacs bug. The fault was caused by the write barrier if the write barrier is currently enabled and the fault happened on a protected page.

In this case, the fault-causing address is added to the `page_fault_table`—an internal data structure that keeps track of all modified pages—by passing it to `vdb_designate_modified`.

The function `unprotect_page_and_mark_dirty` removes the write protection from the page and marks the page dirty by setting a dirty bit associated with the page header. The dirty bit is only used for error checking.

If the fault is not caused by the write barrier, the else branch is executed:

```
  else  /* default SIGSEGV handler */
    {
      char *signal_name;
      if (signum == SIGSEGV)
           signal_name = "SIGSEGV";
      else if (signum == SIGBUS)
        signal_name = "SIGBUS";
      fprintf (stderr,
               "\n\nError: Received %s (%d) for address 0x%x\n",
               signal_name, signum, (int) siginfo->si_addr);
      vdb_remove_signal_handler ();
    }
}
```

First, the signal name is determined, so that the fault handler is able to print a usable error message; then the error message is printed.

To be able to debug XEmacs on such an illegal memory access, the function `vdb_remove_signal_handler` resurrects the previously installed handler and passes the current fault to it. Usually, this is the Lisp backtrace handler that is described in section 3.3.2.

3.3.12 Resume Mark Phase

The mark phase resumes after `gc_cons_incremental_threshold` bytes have been allocated by the client. Then, the function

```
void unprotect_heap_pages (void);
```

removes the write-protection from all formerly protected pages. This turns off the write barrier, and `write_barrier_enabled` is set to false. Next, the function

```
int vdb_read_dirty_bits (void);
```

loops over `page_fault_table` and passes all black objects of the listed pages to the macro

```
gc_write_barrier (obj);
```

that re-colors the objects grey and pushes them onto the mark stack. The number of re-pushed objects is returned by `vdb_read_dirty_bits`.

Afterwards the next mark cycle is started. The maximum of `gc_incremental_traversal_threshold` and `repushed_objects` defines how many traversal steps are executed in the next cycle:

```
repushed_objects = vdb_read_dirty_bits ();
mark_work = (gc_incremental_traversal_threshold
             > repushed_objects) ?
  gc_incremental_traversal_threshold : repushed_objects;
```

The value of `mark_work` is passed to `gc_mark`. This way, in every mark cycle more objects are examined than have been added in the previous write-barrier session. Consequently, the traversal is guaranteed to always terminate. After this mark phase, the traversal may be interrupted again.

3.3.13 Finish Mark Phase

When there are no more grey objects on the mark stack, the traversal is finished. However, as described in section 3.3.8, the root set is not covered by the write barrier. To make sure that the traversal reaches all live objects, all outgoing pointers from the root set have to be traversed again, without further interruption, to examine newly added or modified root set entries.

In most cases, the re-traversal from the roots is fast: Black objects do not have to be traversed again. Measurements in appendix B show that only approximately 3% of the execution time of the mark phase is spent for root-set re-examination. This is included in the execution overhead of the write barrier, which is discussed in detail in section 4.3.3.

This code fragment in `gc_finish_mark` re-marks the root set:

```
if (KKCC_GC_STACK_EMPTY)
  {
    /* Mark root set again and finish up marking atomically. */
    gc_mark_root_set ();
    kkcc_marking (0);
  }
```

Now the heap is fully traversed and the objects are classified in live and garbage. Live objects have their mark bits set to black, all others have white mark bits. The mark phase is finished.

3.3.14 Sweep Phase

The allocator scans the entire heap and frees all white marked objects. The freed memory is recycled and can be re-used for future allocations. The sweep phase is carried out atomically.

3.3.15 Final clean up

First, all variables with respect to garbage collection are reset. The variable `consing_since_gc` that holds the allocated bytes since the last garbage collection is set back to 0, and `gc_in_progress` is set to false.

Finally, `recompute_need_to_garbage_collect` is called to reset the global variable `need_to_garbage_collect`. The garbage collection is finished.

3.3.16 Garbage Collection Control

In the previous sections the several tasks of an incremental garbage collection have been described in detail. This section describes the function

```
void gc (int incremental)
```

that dispatches the control during a garbage collection to the various functions representing the tasks.

The `gc` function is called whenever a garbage collection is invoked, either explicitly by the user or an application, or implicitly because the garbage collection threshold is reached (see section 3.3.5). If it is called with a false argument, garbage collection is carried out in one atomic action. If a garbage collection is already running, it is resumed without any further interruption. A true argument causes `gc` to start or resume an incremental garbage collection. The `gc` function is usually called with the argument `gc_allow_incremental`, see section 3.3.19.

The functions `gc_full` and `gc_incremental` call `gc`:

```
void gc_full (void)
{
  /* never incremental */
  gc (0);
}

void gc_incremental (void)
{
  /* incremental if allow_incremental_gc is set */
  gc (allow_incremental_gc);
}
```

To resume the garbage collection at the right point, the function `gc` maintains a variable `gc_state` that holds the current state of the garbage collection by

storing the last executed task. This way, the dispatcher is always able to call the appropriate function and resume an already running garbage collection correctly. The function is written with one big `switch` statement:

```
switch (gc_state)
  {
  case NONE:              /* no gc running, start a new one */
    gc_state = INIT_GC;
    gc_prepare ();
  case INIT_GC:           /* initialization done */
    gc_state = PUSH_ROOT_SET;
    gc_mark_root_set ();
  case PUSH_ROOT_SET:     /* root set pushed */
    gc_state = MARK;
    gc_mark (incremental);
    if (!KKCC_GC_STACK_EMPTY)
      return;             /* suspend gc */
```

If the mark stack is not empty after the first traversal cycle, an incremental collection is running and the garbage collection has to be suspended. When `gc` is called again, garbage collection resumes here:

```
  case MARK:              /* another mark cycle */
    gc_resume_mark (incremental);
    if (!KKCC_GC_STACK_EMPTY)
      return;             /* suspend gc */
    gc_state = FINISH_MARK;
    gc_finish_mark ();    /* finish mark atomically */
```

More traversal cycles are executed while the mark stack is not empty. Once it is empty, the mark phase is finished atomically and `gc_state` is set accordingly:

```
  case FINISH_MARK:       /* mark done */
    gc_state = FINALIZE;
    gc_finalize ();
  case FINALIZE:          /* finalization done */
    gc_state = SWEEP;
    gc_sweep ();
  case SWEEP:             /* sweep done */
    gc_state = FINISH_GC;
    gc_finish ();
  case FINISH_GC:         /* gc finished */
    gc_state = NONE;
  }
```

The `gc_state` has excess granularity: As only the mark phase is interleaved with client activity, the states `NONE`, `MARK`, and `FINISH_MARK` would be sufficient. But for the collection of statistics and debugging information it is very useful to distinguish all states.

3.3.17 Newly allocated Objects during Garbage Collection

The policy for allocating new objects during a garbage collection affects the amount of floating garbage left behind by an incremental collection algorithm as described in section 2.2.5. This section discusses what color shall be used for allocating new objects when an incremental traversal is running.

If new objects are allocated black or grey, they will survive the garbage collection cycle whether they are still in use by the client or not. This causes floating garbage, if new objects tend to die soon. Additionally, if objects are allocated black, more care has to be taken: As stated above, pages containing black objects have to be covered by the write barrier. Thus, the page with the newly allocated object has to be write-protected. Usually however, new objects get initialized immediately after allocation, and initialization causes heap writes that would immediately be trapped by the write barrier. Subsequently, the new object and all other black objects on that page need to be re-colored grey when the garbage collection is resumed. Consequently, allocating objects black is effectively nothing more than allocating them grey, but requires additional computation.

Allocating new objects grey produces additional traversal work: All new objects have to be examined by the traversal, no matter whether they are really in use by the client or not. If they are in use, their reference would be written into a live object, that in turn would be re-examined by the traversal itself. Allocating a new object grey may result in two objects pushed on the mark stack—a waste of computation and execution time.

The best strategy for allocating new objects is to allocate them white. Not only does this strategy require no special action to be taken by the allocation functions, since all new objects are initially colored white; it is also the most efficient and exact method: New white objects only survive the garbage collection if they are really in use and become part of the reachability graph. They are in use if their reference is written into a live object that either has not been reached by the traversal but will be, or was reached by the traversal and is now colored black and thus covered by the write barrier. In both cases the new object is reached: In the former case the live object is reached when the traversal continues, in the latter case the write barrier catches the update of the black object that subsequently gets re-examined by the traversal. This way the new white object is reached by the traversal.

Consequently, new white objects are reached by the traversal only if they really have to be kept alive. Otherwise they are garbage and die within only one garbage collection cycle.

Conversely, allocating new objects grey or black leads to a averagely five times slower garbage collection execution time than allocating new objects white.

Therefore, the allocator allocates new objects white, and the functions that perform allocation do not have to be modified.

3.3.18 Manual Freeing during Garbage Collection

XEmacs often frees temporary Lisp objects explicitly outside a garbage collection instead of letting the garbage collector do the work. XEmacs uses this

optimization excessively: more objects are freed explicitly outside garbage collection than from the garbage collector, see appendix B for detailed numbers. One example is the code fragment that deals with warnings in `eval` function in `eval.c`:

```
Lisp_Object this_warning_cons = Vpending_warnings;
Lisp_Object this_warning = XCAR (this_warning_cons);
free_cons (this_warning_cons);
class = XCAR (this_warning);
level = XCAR (XCDR (this_warning));
messij = XCAR (XCDR (XCDR (this_warning)));
free_list (this_warning);
```

Here, the `car` of `this_warning_cons` is stored to `this_warning`, then `this_warning_cons` is explicitly freed. The same happens with the list in `this_warning`: The elements of the list are stored in the variables `class`, `level`, and `messij`; then the surround list is manually freed. Objects are only explicitly freed if it is guaranteed that no garbage collection can occur between object generation and object freeing—as in the above code fragment, where `eval` is not called.

The benefit of manual freeing is that memory becomes available for re-use before the next garbage collection. However when a garbage collection is running and an object is manually freed during client activity, special care has to be taken: If the to-be-freed object is marked grey, there is another reference to this object on the mark stack. Additionally to being freed, the reference has to be removed from the mark stack. Otherwise, the traversal will fail because it encounters an invalid reference: The memory pointed to may still be unused or it may already contain a newly allocated object. Both cases can cause the traversal to fail.

Removing an object from the mark stack is very costly: the entire stack has to be scanned to find the object. Therefore, manual calls to the various free functions for Lisp objects are ignored when a garbage collection is suspended. This causes memory overhead, but only until the running garbage collection is finished and the heap is swept. In that case, the objects are collected anyway because temporary objects are not reachable and have not been marked. Measurings of the memory usage and overhead can be found in section 4.3.4.

3.3.19 Lisp Interface

The new garbage collector can be accessed directly from Emacs Lisp. Basically, two functions invoke the garbage collector:

`(gc-full)` starts a full garbage collection. If an incremental garbage collection is already running, it is finished without further interruption. This function guarantees that unused objects have been freed when it returns.

`(gc-incremental)` starts an incremental garbage collection. If an incremental garbage collection is already running, the next cycle of incremental traversal is started. The garbage collection is finished if the traversal completes. Note that this function does not necessarily free any memory. It only guarantees that the traversal of the heap makes progress.

The old garbage collector uses the function `(garbage-collect)` to invoke a garbage collection. This function is still in use by some applications that explicitly want to invoke a garbage collection. Since these applications may expect that unused memory has really been freed when `(garbage-collect)` returns, it maps to `(gc-full)`.

The new garbage collector is highly customizable during runtime; it can even be switched back to the traditional mark-and-sweep garbage collector: The variable `allow-incremental-gc` controls whether garbage collections may be interrupted or if they have to be carried out in one atomic action. Setting `allow-incremental-gc` to `nil` prevents incremental garbage collection, and the garbage collector then only does full collects, even if `(gc-incremental)` is called. Non-`nil` allows incremental garbage collection.

This way applications can freely decide what garbage collection algorithm is best for the upcoming memory usage. How frequently a garbage collection occurs and how much traversal work is done in one incremental cycle can also be modified during runtime:

As seen in section 3.3.5, the number of bytes allocated since the last garbage collection determines when the next garbage collection runs. The variable `gc-cons-threshold` holds the number of bytes that have to be allocated before the next garbage collection starts. The value of `gc-cons-threshold` is only used after a garbage collection has completed. When an incremental garbage collection is interrupted, the variable `gc-cons-incremental-threshold` determines when the next cycle has to be invoked. How many bytes have been allocated since the last full garbage collection or the last garbage collection cycle, can be queried by calling the function `(consing-since-gc)`.

The duration of the interruption by the garbage collector can also be adjusted: The variable `gc-incremental-traversal-threshold` holds the number of elements that are processed in one cycle of incremental traversal.

The following table shows the default values of the threshold variables:

Short Description and Variable Name	**Default Value**
Bytes allocated between garbage collections `gc-cons-threshold`	1,000,000 bytes
Bytes allocated between garbage collection cycles `gc-cons-incremental-threshold`	200,000 bytes
Elements processed in one incremental traversal cycle `gc-incremental-traversal-threshold`	100,000 elements

These default values provide good performance and are based on my experiences and some trial-and-error tests. A future task is to write an application that runs tests with different threshold settings to determine the best default configuration for each system.

By changing the values of these variables, applications can influence the frequency and execution times of garbage collections. For example, real-time applications may decide to postpone garbage collection until they have finished. The user can set these options through XEmacs's customization interface that groups settings together and provides an easy-to-use interface [SW]. I added all the options to the "Storage Allocation Group" that holds all allocator and garbage collector related settings. Customization of the collector's options is invoked by typing

```
M-x customize RET alloc RET
```

This way, the user can easily experiment with the new incremental garbage collector and adjust it to her needs.

Chapter 4

Results

This chapter shows the results of the work on the new garbage collector. First, the progress in modularizing XEmacs and increasing the client reactivity is shown. Later, I support my statements by providing results of performance and memory usage measurements.

4.1 Modularization

Information hiding, encapsulation, and abstraction are basic concepts of modern software development; they all describe different views of one idea: hiding the design decisions in a computer program that are likely to change. Required information is only accessible through a stable interface that protects other parts of the underlying design.

XEmacs's old memory manager did not conform to these basic concepts in all components: it had many dependencies with other, unrelated parts of XEmacs. The memory manager's close coupling to the client hinders program development. By contrast, the new memory manager is fully separated from the client and consists of three modules that are independent from each other; they provide basic functionality through a well-defined interface that is described in appendix A. Here are short descriptions of the modules:

Allocator: The allocator provides memory allocation and free functions to the client, supports the garbage collector's sweep phase and provides heap protection mechanisms for the write barrier that work on pages.

Garbage Collector: The garbage collector provides functionality for the client: The client communicates the roots of accessibility and can invoke a garbage collection. The garbage collector receives information about modified objects from the write barrier.

Write Barrier: The write barrier provides the platform-dependent memory-protection mechanisms to be used by the allocator and the platform-dependent fault handlers to inform the garbage collector about modified objects. No interface between the client and the write barrier is needed.

Figure 4.1 shows a scheme of the memory manager's modules connected by lines indicating the interaction between the components. The full interface

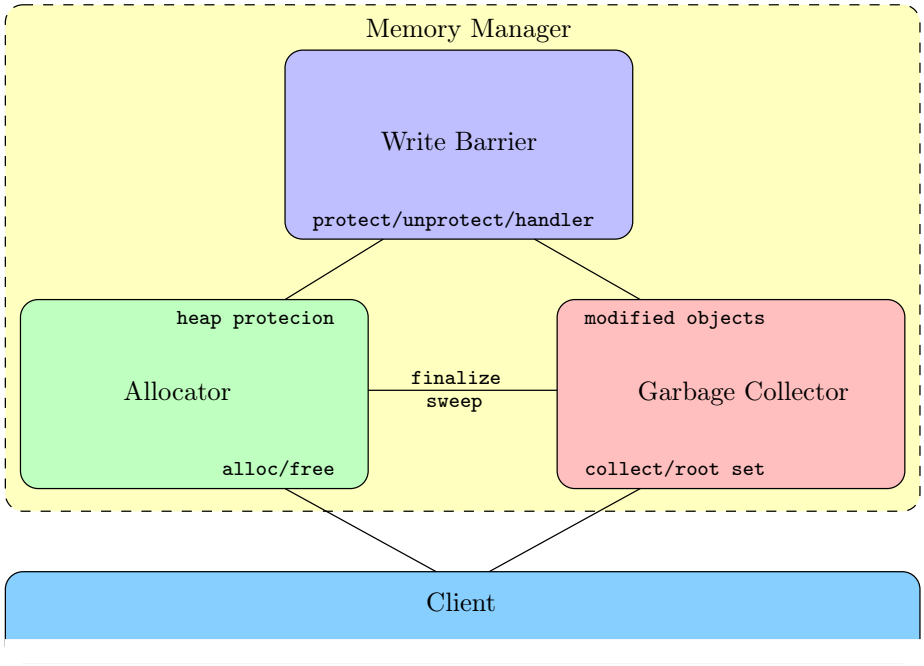

Figure 4.1: Modules of the memory manager

specification is listed in Appendix A.

Future changes and improvements to parts of the memory manager can be accomplished more easily with the strict modularization.

4.2 Reactivity

Another result from using the new garbage collector is the increased reactivity of the client. Figure 4.2 points out the difference between non-reactive traditional mark-and-sweep collectors and the new incremental one with increased reactivity.

In the figure, the lower bars indicate client activity, while the upper bars stand for memory manager activity. In the traditional scheme, garbage collection is carried out as one atomic action, with no client activity while the collector runs. Conversely, the incremental garbage collector stops repeatedly to give the client execution time, and the client can resume whatever task it may have pending. The pause times caused by the incremental collector are smaller, and reactivity is increased.

As stated in section 2.1.3, it is hard to measure the performance of garbage collection algorithms. It is especially difficult to express the interactive reactivity of an incremental garbage collector in objective numbers and figures. Reactivity is rather a subjective improvement that the user experiences by using XEmacs for her daily work. However, the average client pause times and the general performance of the collector give some indication, as evaluated in section 4.3.3.

Instead of being annoyed by the garbage collector's interruptions, ideally the user does not notice garbage collection at all. Of course, the garbage collection

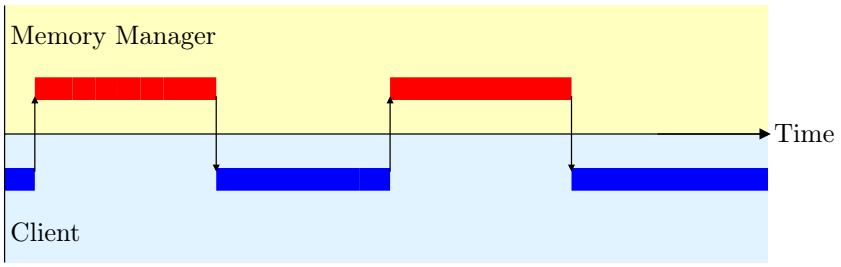

Figure 4.2.1: Traditional mark-and-sweep execution

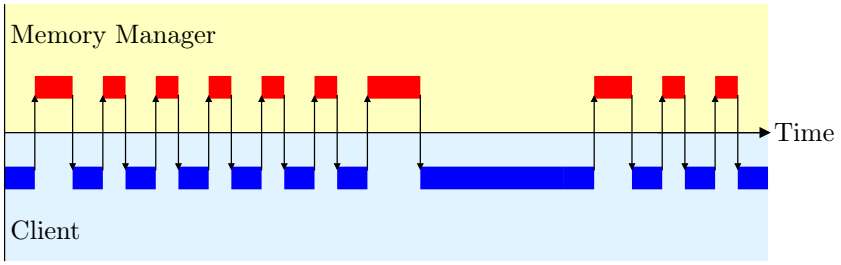

Figure 4.2.2: Incremental mark-and-sweep execution

Figure 4.2: Comparison of traditional and incremental mark-and-sweep garbage collection execution activity

still has to do its job and make garbage memory re-usable again. Garbage collection has to occur unnoticed, but still provide a reasonable sized heap. How much memory the different schemes are using is measured in section 4.3.4.

4.3 Measuring Performance and Memory Usage

This section measures how the new incremental garbage collector influences performance and memory usage of XEmacs. First, the test cases and measuring conditions are introduced; then, the results are discussed.

4.3.1 Standard XEmacs Usage Pattern

Shane Holden and Steve Baur wrote a benchmark that is part of the XEmacs packages distribution, called `bench.el`. It provides a good basis for measuring XEmacs's performance, as well as verifying that XEmacs works as expected. Here is a list of the tests:

(1) Towers of Hanoi: Solve the Towers of Hanoi game and show the solution in an animation.

(2) Font Lock: Open a big file containing Emacs Lisp source code—the source of the XEmacs news reader `gnus.el`—and color the source code according to its syntax.

(3) Large File Scrolling: Scroll the whole file by moving the cursor—*point* in XEmacs terms—forward line by line.

(4) Frame Creation: Create new frames, wait until they come up, and destroy them again.

(5) Generate Words: Generate 10,000 words containing less than 10 random letters.

(6) Sort Buffer: Sort the buffer that contains the previously generated random words alphabetically.

(7) Large File Bytecode Compilation: Compile the big file used in phase (1) to bytecode.

(8) Text Insertion: Insert the string "`0123456789\n`" 100,000 times into an empty buffer.

(9) Loop Computation: Run a loop that increments a counter.

(10) Make a Few Large Size Lists: Generate 10 large lists, each containing 1,000,000 elements. Immediately after generation these lists become garbage.

(11) Garbage Collection Large Size Lists: A full garbage collection is run to collect the garbage lists generated in the previous step.

(12) Make Several Small Size Lists: Generate 1,000,000 small lists, each containing 10 elements. Immediately after generation these lists become garbage.

(13) Garbage Collection Small Size Lists: A full garbage collection is run to collect the garbage lists generated in the previous step.

The benchmark provides standard XEmacs usage with the tasks (1)–(9), and extensive testing of the memory manager with (10)–(13).

Having both performance and memory manager test cases, `bench.el` provides good measurable results for standard XEmacs usage with a focus on garbage collection issues. Apart from the standard XEmacs user pattern, other measurements provide interesting results as well:

Startup speed: Starting an Editor should be fast. Do the modifications to the memory manager alter startup performance?

Regression tests: `make check` provides an auto-testing facility for XEmacs, to test if XEmacs behaves correctly. The tests are referred to as *regression tests*, to allow developers to check if their changes lead to any unexpected regressions in other parts of XEmacs [WTB+]. These tests provide a log of passed and failed tests, which allow the developer to investigate the source of the error and fix the bug. Every important feature of XEmacs has its own test cases. Since these tests are pushing XEmacs to its limits, they provide interesting results.

The results of these measurements are listed in appendix B.

The next section defines the conditions under which the measurements take place.

4.3.2 Measuring Conditions

The measurements are made with a MULE-enabled[1] XEmacs 21.5 beta 21. All tests are run with and without the described incremental garbage collector enabled. In both cases, XEmacs is compiled in developer mode, which means no compiler optimization[2] and all error checking and debugging information enabled. Note that collection and output of measuring data slows down XEmacs additionally.

The tests are run on a machine equipped like this:

- Intel Pentium 4 with 3,0 GHz and 1 GB memory
- Linux Kernel 2.6.11
- gcc 3.3.5

All tests are run with a *vanilla* XEmacs—that is an XEmacs that does not load any user specific configuration files. Starting XEmacs with the command-line argument '`-vanilla`' brings up a plain XEmacs without any user-side extensions. This way no user setting can influence the results.

The test cases are run with the default values for the threshold variables, as listed in section 3.3.19.

4.3.3 Performance Results

The collected information during the performance measurements brought up the following results:

Performance	Total Time	Time in GC	Number of GCs	Number of Cycles	Time per GC	Time per Cycle
non-incremental	82.93 s	19.76 s	61	61	324 ms	324 ms
incremental	78.99 s	15.62 s	43	132	363 ms	118 ms

First of all it is interesting, that the total execution time of the bench mark test was approximately 4 seconds shorter with the incremental collector enabled than with the non-incremental one. This is not caused by a faster garbage collector—the incremental collector cannot be faster as it does the same work only incrementally. The reason is that with the incremental collector less complete collections are run than with the non-incremental collector. Figure 4.2 illustrates this fact: the non-incremental traditional mark-and-sweep collector makes two complete garbage collections, whereas the incremental collector achieves less than one-and-a-half, because it allows more client activity.

The measured times and numbers reflect this: The time spent in the non-incremental garbage collector is 19.76 seconds; in that time, 61 complete garbage collections run. The incremental garbage collector executes only 43 complete collections in 15.62 seconds. But the incremental collector splits these 43 collections into 132 single collection cycles. This means that one complete collection consists averagely of three cycles—the client runs twice during one complete garbage collection.

[1] MULE: multilingual environment, support for input handling and display of multilingual text

[2] XEmacs runs approximatelly 20%–25% faster with optimization (`-O2`)

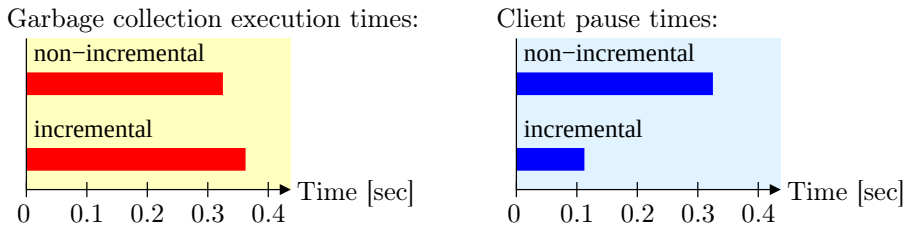

Figure 4.3: Comparison of garbage collection and client pause times

As expected, the time of one incremental collection (363 milliseconds) is slower than a non-incremental collection (324 milliseconds). The overhead is about 12%; it is due to re-examining objects twice that have been caught by the write barrier.

The average time per garbage-collection cycle shows the benefit of the incremental collection: The time per garbage collection corresponds to the time the client is interrupted. One cycle with the incremental collector takes only 118 milliseconds and improves the traditional collector's 324 milliseconds by more than 63%. The incremental collector decreases client pause times by two-thirds on average.

Figure 4.3 illustrates comparisons of average garbage collection execution times and average time per cycle which corresponds to the average client pause time.

Figure 4.4 shows garbage collection and client interaction during bench mark execution for incremental and non-incremental collection schemes.

The most noticeable client pause caused by the incremental garbage collector happens at approximately 67 seconds into the execution. This is the time when

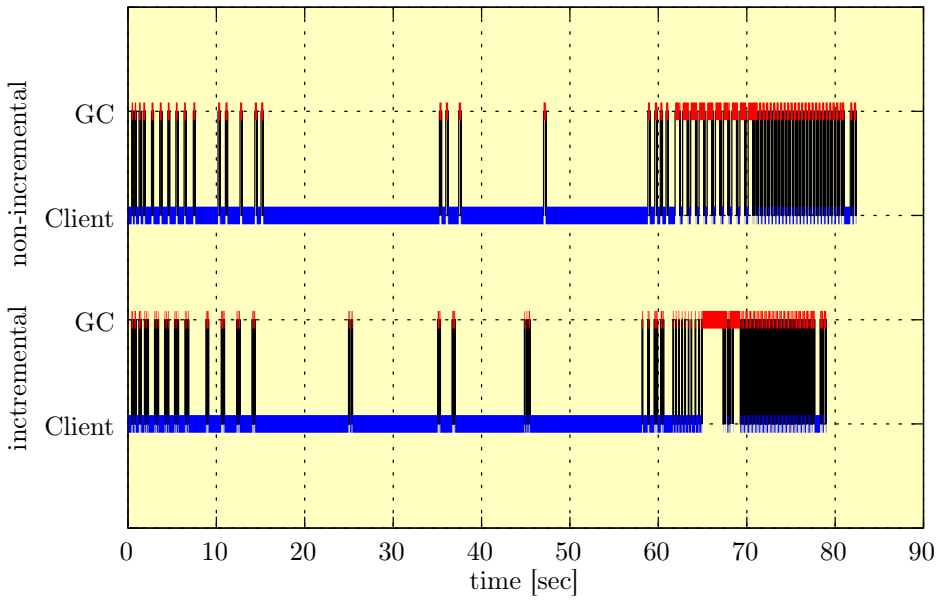

Figure 4.4: Garbage collection times comparison

the large-list benchmark (11) is finished and a full garbage collection (12) is forced.

Since incremental collections run during the generation of the large lists, the collector treats some of these lists as live—they are floating garbage. Thus, they are marked by the non-interruptible traversal, which causes the long pause time. The incremental garbage collector performs badly for the large-list benchmark: The client is interrupted for a long time. Additionally, there is temporary memory overhead caused by floating garbage. This is discussed in the next section.

When the benchmark pushes the limits of the garbage collector with the special large-list test, the incremental garbage collector shows its disadvantages. Fortunately, the large-list test is not a typical scenario. Additionally, the benchmark forces a full garbage collection. Without this, these lists are marked and collected after a few incremental cycles, that do not cause noticeable pause times, as my measurements show:

Performance without full GC	Total Time	Time in GC	Number of GCs	Number of Cycles	Time per GC	Time per Cycle
incremental	78.22 s	14.72 s	36	152	409 ms	96 ms

Without the forced full garbage collection, the average client pause time reduces to 96 milliseconds, again approximately 19% less pause time than with the incremental collector with forced full collections. And the average memory usage only grows to 45.02 MB, and increases by only 1% compared to the average memory usage of the incremental collector with the full collections that the next section describes in detail.

The other tasks of the benchmark are all solved successfully: The average client pause time is reduced to one-third, with the long pause time during large-list collection included.

4.3.4 Memory Usage Results

Memory measurements of the Lisp heap and of the virtual memory of the process brought the following results:

Memory Usage	Heap		Virtual Memory	
	average	maximum	average	maximum
non-incremental	7.68 MB	17.34 MB	54.43 MB	79.41 MB
incremental	11.64 MB	63.74 MB	44.37 MB	97.71 MB

The heap managed by the incremental collector is, on average, 52% bigger than the non-incremental collector's heap, and the maximum heap usage is nearly three times as big. To analyze the memory usage I provide plots that show the development of the memory usage over time during execution: Figure 4.5 shows the memory growth of the non-incremental collector, figure 4.6 illustrates the incremental collector's memory occupancy. In both figures, the upper line indicates the virtual memory usage, the lower line the heap usage. The bars along the x-axis indicate garbage collection activity. The correlation between garbage collection and heap usage is easily observable: Garbage collection causes the heap to shrink.

The first conspicuous item on both figures is the peek of the virtual memory up to 80 MB at approximately 8 seconds into the execution. This peek is

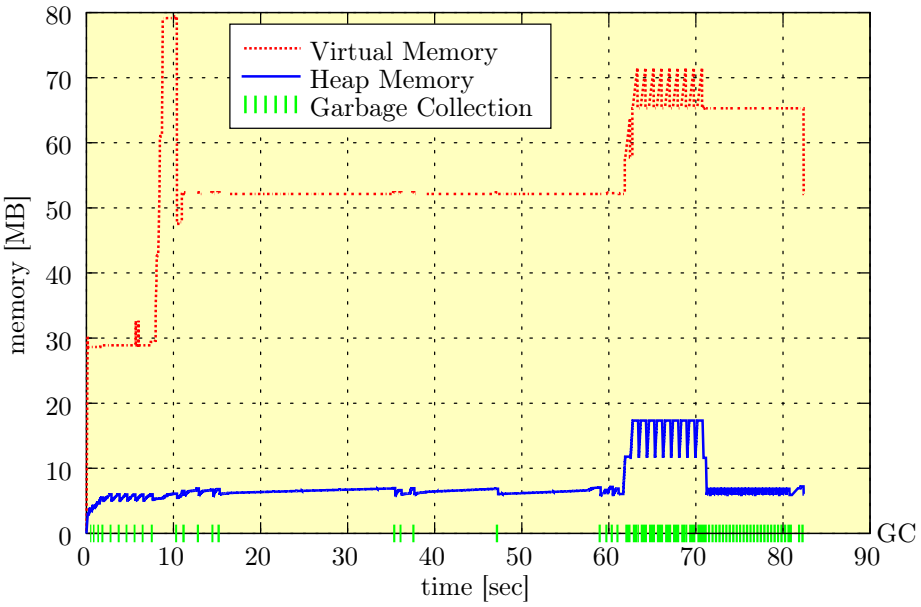

Figure 4.5: Memory usage of non-incremental garbage collection

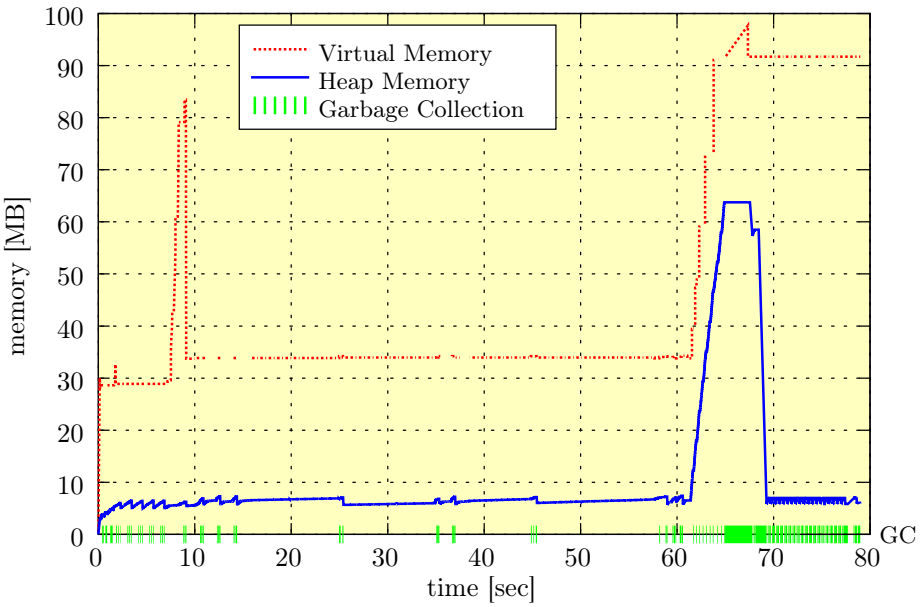

Figure 4.6: Memory usage of incremental garbage collection

not caused by the Lisp engine—the heap does not grow. The peek happens concurrently when the XEmacs frame pops up on the screen. This indicates that the peek is caused by generating the graphical user interface and connecting to the X Server that displays the frame. It has nothing to do with the garbage collector.

The measurements for average virtual memory usage show that the incremental collector uses on average 18% less memory than the non-incremental one. I do not know why the incremental collector provides a better virtual memory usage than the non-incremental does.

Although it is not easy to understand the virtual memory usage, it is still interesting to take a look at the virtual memory: The memory manager's allocator can cause fragmentation. Therefore, it is important to keep an eye on the virtual memory: does it shrink when the heap shrinks?

The growth of the heap also grows the virtual memory. This happens 60 seconds into the execution, when the large-list benchmark is run. The plot of the non-incremental garbage collector shows how the generation of the lists grow the heap and virtual memory, and how garbage collection shrinks both. The virtual heap of the incremental collector does not shrink by the same amount the heap does—the allocator wastes memory due to fragmentation. Fixing this is on my future work list in section 5.1.

As described for the performance measurements, the large-list benchmark shows that the incremental scheme can leave a lot of floating garbage behind. When the benchmark generates the the first list, an incremental garbage collection cycle is started, because the threshold is exceeded. At that time, the list is in use—the garbage collector marks it live. When the benchmark is finished with the first list and discards it, the list is already part of the traversal and is still kept alive. This happens repeatedly during the large-list benchmark, and eventually causes the large heap.

Figure 4.7 shows a direct comparison of the incremental and non-incremental heaps. Aside from the large-list benchmark there is no big difference between them. For "normal" test cases that do not push the limits as the large-list test

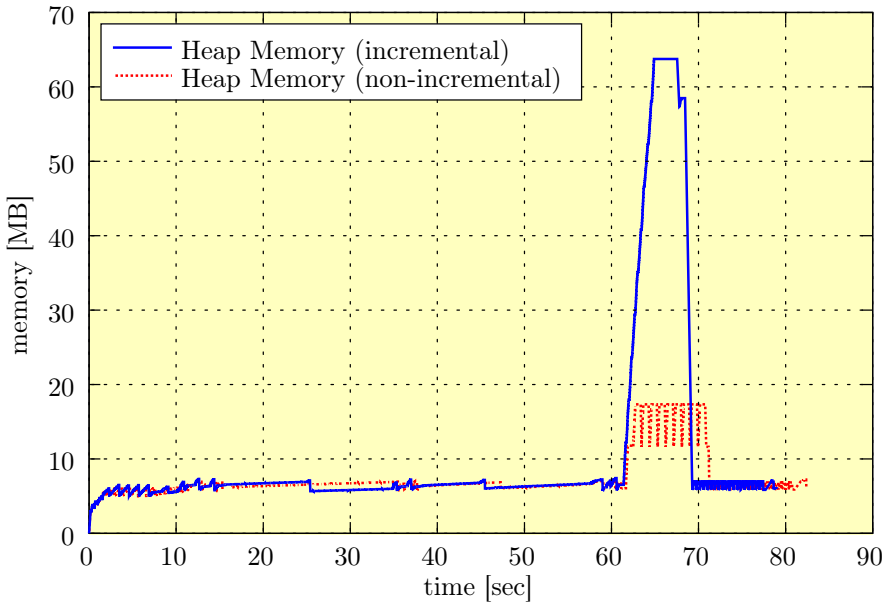

Figure 4.7: Memory usage of incremental and non-incremental garbage collection

does, there is no significant memory overhead, neither caused by floating garbage, nor by the smaller number of complete garbage collections. The memory overhead is large only in extreme cases.

4.3.5 Discussion of Results

The memory overhead of the incremental garbage collector seems large looking at the 52% increase of average memory usage. But having a closer look at figure 4.7 shows that the memory usage is approximately the same during the most of the program execution, except for the large-list benchmark. The large-list benchmark reveals the weakness of an incremental algorithm: When large amounts of objects are allocated, the incremental garbage collector and the client outrun each other. Thus, the incremental collector only achieves poor performance and incurs memory overhead.

To solve these problems, special treatment of such situations has to be implemented in the garbage collector. Another solution to this problem exists on the application side: When the application plans to allocate large amounts of memory but does not need it very long, it can modify the garbage collector's behavior by changing the threshold variables as described in section 3.3.19.

Aside from the measurings, the memory usage I observe by using XEmacs with the new incremental garbage collector enabled on a day-to-day basis is not significantly different than before, with the traditional collection scheme enabled. However the reduced client pause times stand out positively.

Especially in the benchmark scrolling test one can easily see the difference: The scrolling with the non-incremental collector seems to be stuck several times for a few moments, whereas scrolling goes fluently with the incremental collector.

The results regarding performance and memory usage are promising. Especially the client pause times which have been reduced to one-third are a big improvement.

Chapter 5

Conclusion

The new incremental garbage collector makes XEmacs more reactive by interleaving small amounts of collection work with program execution. Garbage collection no longer interrupts the user's work, and using XEmacs feels smoother.

However, there is a price to pay: the mutator and the collector have to be coordinated using a write barrier. In particular, storing pointers into objects when the collector is suspended is more costly, and so is the repeated traversal of modified objects. Consequently, a full garbage collection cycle is slightly slower than before.

Additionally, the memory overhead of the incremental garbage collector is bigger due to floating garbage, because garbage objects may go unreclaimed until the next cycle. But measurements show that the memory usage is still within reasonable boundaries, and the amount of floating garbage is typically small.

Trading slower garbage collection and higher memory usage for better user interaction and increased program reactivity is worth the price. Smooth user interaction is crucial for interactive programs. XEmacs now meets the demands: Client pause times caused by garbage collections have been reduced to one-third. And there is still potential for improvement: The incremental garbage collector was created with the focus on correctness rather than optimization. Consequently, I expect performance improvements by turning the the focus to optimization.

Along with the changes to XEmacs's garbage collector I have modularized the memory manager. Tight coupling of the memory manager and the client is removed, so that future projects in this area can be accomplished easier and faster.

This work shows, that the implementation of a write barrier is highly platform-dependent. Currently ports exist to various UNIX/Linux systems, Mac OS X, Cygwin, and native Windows.

The first real-life test for my new incremental garbage collector is coming up: Within the next few weeks I am going to commit my changes to the XEmacs source repository. Once published, the incremental garbage collector will be tested on a variety of systems and platforms with different configurations by the XEmacs developers and the XEmacs community. With the resulting feedback I can fix bugs, advance the incremental garbage collector, and port the write barrier to more systems.

Finally, the achievement of my work can be summarized in on sentence: The new incremental garbage collector provides a fast and extensible automatic memory management for XEmacs.

5.1 Future Work

I have worked a lot on XEmacs's memory manager so far, but there are still some projects left to take care of:

Finalization of Dead Objects
Finalization is the name of a technique to apply actions to an object when the object is no longer in use by the client. An object has a function called *finalizer* that runs after the garbage collector found out that the object is garbage.

Currently, XEmacs's finalizers run synchronously during a garbage collection, just before the sweep phase. Since the finalizers run while still in the middle of the collection, special care has to be taken how finalizers are written and what actions they take. This can easily cause errors.

Therefore, it is better to have the finalizers run asynchronously after the garbage collection. This is how finalization is implemented by the Boehm-Demers-Weiser conservative garbage collector [Boe] and most other implementations described by Jones and Lins [JL96]. XEmacs already has an asynchronous finalization strategy, currently only used for one Lisp object type: ephemeron [Hay97]. The finalizers of ephemerons run after the garbage collector. XEmacs needs to use this finalization strategy for all Lisp objects.

Optimize the Allocator
The allocator is sometimes wasting memory due to fragmentation. The allocator can be optimized to tap its full potential. My previous work [Cre04] describes opportunities for optimizing the allocator.

Speedup Mark Phase
Performing the traversal of live objects is extremely costly with XEmacs's mark algorithm. One approach to speed it up is to treat the most common objects specially, like cons cells. This way, the traversal algorithm would not have to parse frequent objects, as their treatment would be hard-coded.

Reingruber [Rei01] worked on a different approach: He re-designed the layout of the objects to separate their pointer-containing cells from no-pointer opaque data. This way, the traversal would have no need to parse a object, it can assume that every cell in the pointer part of the object has to be examined. This speeds up the mark phase but is a big project.

Generational Garbage Collector
Although the incremental garbage collector outruns XEmacs's traditional mark-and-sweep garbage collector in client reactivity, incremental garbage collectors still suffer from a number of drawbacks: Since all active data must be marked in each garbage collection, a collection cycle takes very long.

> Generational garbage collectors assume that younger objects are likely to die soon, whereas older objects are likely to survive through many collections. Generational collectors avoid to mark older objects repeatedly by segregating objects into multiple areas by age, and scavenging areas containing older objects less often than younger ones. The separation of objects leads to faster garbage collections.

I will stay with the XEmacs community and keep on working on the memory manager.

Appendix A

Interface to the Memory Manager

A.1 Interface to the Allocator

Internal Allocator Functions

The internal allocator functions are the low-level interface to the allocator. The client usually does not call these directly.

`void init_mc_allocator (void)` [Function]

Initialize the allocator. This has to be called prior to requesting memory.

`void *mc_alloc (size_t size)` [Function]

Allocate a block of memory of given `size` and return the pointer to it.

`void *mc_alloc_array (size_t size, int elemcount)` [Function]

Allocate a block of memory as an array with `elemcount` elements of given `size` and return the pointer to it. Arrays contain several objects that are allocated in one consecutive block of memory with each element being a fully qualified object—that is, it has a Lisp object header and a mark bit. Objects like hash tables and dynamic arrays use this function.

`void *mc_free (void *ptr)` [Function]

Free the object pointed to by `ptr` and make its memory re-usable again. The memory must have been returned by a previous call to `mc_alloc()`. This can be used to free memory explicitly, outside a garbage collection.

`void *mc_realloc (void *ptr, size_t size)` [Function]

Modify the size of the memory block pointed to by `ptr`. Return the address of the new block of given `size`. The content of the memory block will be unchanged to the minimum of the old and new sizes: if the new size is smaller, the overlaying data is cut off; if the new size is bigger, the newly allocated memory will be uninitialized.

`void *mc_realloc_array (void *ptr, size_t size, int elemcount)` [Function]

Modify the size of the array pointed to by `ptr`. Return the address of the new array block with `elemcount` elements of given `size`. The content of the memory block will be unchanged to the minimum of the old and new sizes: if the new size is smaller, the overlaying data is cut off; if the new size is bigger, the newly allocated memory will be uninitialized.

`EMACS_INT mc_get_page_size (void)` [Function]

Return the `PAGESIZE` the allocator uses. Generally equals to the system's `PAGESIZE`.

Client Interface

Functions the client uses to allocate the Lisp objects:

`alloc_lrecord_type (type, lrecord_implementation)` [Macro]

Allocate a Lisp object of given `type` and initialize it with lrecord implementation `imp`. Use this macro if the size of the Lisp object is fixed, which means it is equal to the size of its struct.

`void *alloc_lrecord (Bytecount size, const struct lrecord_implementation *imp)` [Function]

Allocate `size` bytes for a Lisp object and initialize it with lrecord implementation `imp`. Use this function if the Lisp objects size varies—it is not equal to the size of its struct.

`void *alloc_lrecord_array (Bytecount size, int elemcount, const struct lrecord_implementation *imp)` [Function]

Allocate an array of `elemcount` Lisp objects of size `size` and initialize every object in this array with lrecord implementation `imp`.

`void free_lrecord (Lisp_Object rec)` [Function]

Free the Lisp object explicitly. This can be used to free memory outside a garbage collection.

Garbage Collection Support

Functions the allocator provides for the garbage collector:

`void mc_finalize (void)` [Function]

Scan the entire heap and finalize all unmarked objects that have a finalizer.

`void mc_sweep (void)` [Function]

Scan the entire heap and free all unmarked objects.

Dumper Support

Functions the allocator provides for the portable dumper:

`void mc_finalize_for_disksave (void)` [Function]

Scan the entire heap and finalize all unmarked object that have a finalizer for disk save. Used by the dumper to keep the dump image as small as possible.

Mark Bits Interface

Functions the allocator provides to the garbage collector for dealing with mark bits:

`void set_mark_bit (void *ptr, EMACS_INT value)` [Function]

Set the mark bit of the object pointed to by `ptr` to `value`.

`EMACS_INT get_mark_bit (void *ptr)` [Function]

Return the mark bit of the object pointed to by `ptr`.

`MARK (ptr)` [Macro]

Mark the object pointed to by `ptr`. Set the mark bit to `BLACK`.

`UNMARK (ptr)` [Macro]

Unmark the object pointed to by `ptr`. Set the mark bit to `WHITE`.

`MARKED_P (ptr)` [Macro]

Evaluate to true if the mark bit is set (mark bit $\neq$ `WHITE`).

`MARK_{WHITE|GREY|BLACK} (ptr)` [Macro]

Mark the object pointed to by `ptr` `WHITE`|`GREY`|`BLACK`.

`MARKED_{WHITE|GREY|BLACK}_P (ptr)` [Macro]

Evaluate to true if the mark bit is set to `WHITE`|`GREY`|`BLACK`.

Write Barrier Support

Functions the allocator provides for the write barrier:

`void protect_heap_pages (void)` [Function]

Scan the entire heap and write protect all pages that contain `BLACK` objects.

`void unprotect_heap_pages (void)` [Function]

Scan the entire heap and remove write protection for all pages.

`void unprotect_page_and_mark_dirty (void *ptr)` [Function]

Remove write protection from the page `ptr` points to and set its dirty bit.

`int repush_all_objects_on_page (void *ptr)` [Function]

Scan the dirty page `ptr` points to and push all `BLACK` marked objects on the mark stack for re-examination by the traversal. Return how many objects have to be re-examined by the garbage collector.

A.2 Interface to the Write Barrier

Platform-Dependent

Every platform-dependent version of the virtual-dirty-bit write barrier resides in its own source code file. Files that currently exist are: `vdb-posix.c` for POSIX-compliant platforms, `vdb-darwin.c` for Mac OS X, `vdb-win32.c` for Cygwin and native Windows, and `vdb-fake.c` that holds a fake implementation for platforms that currently do not support the virtual-dirty-bit write barrier. This fall-back "fake" implementation turns of the incremental write barrier at runtime and does not allow any incremental collection. The garbage collector then acts like a tradtional mark-and-sweep collector.

`void vdb_install_signal_handler (void)` [Function]

Install the platform-dependent signal handler.

`void vdb_protect (void *ptr, EMACS_INT len)` [Function]

Set the platform-dependent memory protection for the memory region [`ptr`, `ptr+len`[.

`void vdb_unprotect (void *ptr, EMACS_INT len)` [Function]

Remove the platform-dependent memory protection for the memory region [`ptr`, `ptr+len`[.

Platform-Independent

The platform independent part of the virtual dirty bit write barrier is in `vdb.c`.

`void vdb_start_dirty_bits_recording (void)` [Function]

Start the write barrier. This function is called when a garbage collection is suspended and the client is resumed.

`void vdb_stop_dirty_bits_recording (void)` [Function]

Stop the write barrier. This function is called when the client is suspended and the garbage collection is resumed.

`void vdb_designate_modified (void *addr)` [Function]

Add the object pointed to by `addr` to the write barrier's internal data structure that stores modified objects. This function is called by the write barrier's fault handler.

`void vdb_read_dirty_bits (void)` [Function]

Read out the write barrier's internal data structure that stores modified objects and pass the information to the garbage collector. This function is called by `vdb_stop_dirty_bits_recording()`. Return how many objects have to be re-examined by the garbage collector.

A.3 Interface to the Garbage Collector

KKCC mark functions

Functions that manage the KKCC mark stack:

```
void kkcc_gc_stack_push (void *ptr,                          [Function]
                         const struct memory description *desc)
```

Push the Lisp object pointed to by `ptr` and its memory layout description `desc` onto the mark stack.

`kkcc_gc_stack_entry *kkcc_gc_stack_pop (void)` [Function]

Pop a `kkcc_gc_stack_entry` from the mark stack. A `kkcc_gc_stack_entry` consists of two fields: `ptr` contains the address of the object and `desc` contains the object's memory layout description.

Internal Functions

The internal garbage collector functions are the low level interface to the allocator. The client usually does not call these directly.

`void gc_start (void)` [Function]

Start a full garbage collection, initialize needed data structures.

`void gc_mark_root_set (void)` [Function]

Initialize the mark stack and mark the root set.

`void gc_mark (int incremental)` [Function]

Proceed `incremental` steps of traversal work. If `incremental` is ≤ 0 do not mark incrementally, but process the remaining traversal atomically.

`void gc_suspend_mark_phase (int incremental)` [Function]

Suspend the mark phase, start the write barrier, and return control to the client.

`int gc_resume_mark_phase (int incremental)` [Function]

Resume the mark phase by stopping the write barrier and reading out the write barrier's data. Return how many objects have to be re-examined by the garbage collector.

`void gc_finish_mark (void)` [Function]

Finish the traversal by wrapping up the mark phase without further interruption.

`void gc_finalize (void)` [Function]

Scan the entire heap and finalize all unmarked objects that have a finalizer.

`void gc_sweep (void)` [Function]

Scan the entire heap and free all unmarked objects.

`void gc_finish (void)` [Function]

Finish garbage collection, free used data structures.

`void gc (int incremental)` [Function]

Main garbage collection function that dispatches to the several garbage collection functions mentioned above. Maintains a variable that keeps the state of the current collection. The parameter `incremental` determines if the collection is carried out incrementally or atomically.

Client Interface

Functions the client uses to invoke garbage collections:

`void gc_full (void)` [Function]

Perform a full garbage collection without interruption. If an incremental garbage collection is already running it is completed without further interruption. This function calls `gc()` with a negative or zero argument.

`void gc_incremental (void)` [Function]

This function starts an incremental garbage collection. If an incremental garbage collection is already running, the next cycle of traversal work is done, or the garbage collection is completed when no more traversal work has to be done. This function calls `gc` with a positive argument, indicating how many objects can be traversed in this cycle.

`void recompute_need_to_garbage_collect (void)` [Function]

Determines if a certain threshold—see section A.4—is reached and if a garbage collection has to be performed. This function is called whenever `Feval` is executed.

Root Set Interface

The client uses the root set interface to inform the garbage collector about the roots of accessibility.

`void staticpro (Lisp_Object *obj)` [Function]

Add static C variables containing Lisp objects to the root set.

`void mcpro (Lisp_Object obj)` [Function]

Add a Lisp object to the root set.

`GCPRO (), UNGCPRO ()` [Macro]

Add/remove local variables containing Lisp objects to/from the root set.

Write Barrier Interface

The garbage collector has one function to receive notice about modified objects from the write barrier:

`gc_write_barrier (obj)` [Macro]

Inform the garbage collector that the Lisp object `obj` has been modified by the client. This function is called by `vdb_read_dirty_bits()`. This function pushes the objects on the mark stack for re-examination by the traversal.

A.4 Lisp Interface

These functions and variables can be accessed from Emacs Lisp:

`(gc-full)` [Function]

This function performs a full garbage collection. If an incremental garbage collection is already running, it completes without any further interruption. This function guarantees that unused objects are freed when it returns. Garbage collection happens automatically if the client allocates more than '`gc-cons-threshold`' bytes of Lisp data since the previous garbage collection.

`(gc-incremental)` [Function]

This function starts an incremental garbage collection. If an incremental garbage collection is already running, the next cycle starts. Note that this function has not necessarily freed any memory when it returns. This function only guarantees, that the traversal of the heap makes progress. The next cycle of incremental garbage collection happens automatically if the client allocates more than '`gc-incremental-cons-threshold`' bytes of Lisp data since previous garbage collection.

`(garbage-collect)` [Function]

The traditional garbage collector uses this function to invoke a garbage collection. It starts a full garbage collection; it redirects to `(gc-full)`, to maintain backwards compatibility for programs that assume that the heap is really swept after invoking this function.

`allow-incremental-gc` [Variable]

Non-nil means to allow incremental garbage collection. Nil prevents incremental garbage collection, the garbage collector then only does full collects (even if `(gc-incremental)` is called).

`gc-cons-threshold` [Variable]

Number of bytes of consing between garbage collections. Garbage collection can happen automatically once this many bytes have been allocated since the last garbage collection. All data types count.

`gc-cons-incremental-threshold` [Variable]

Number of bytes of consing between cycles of incremental garbage collections. The next garbage collection cycle can happen automatically once this many bytes have been allocated since the last garbage collection cycle. All data types count.

`(consing-since-gc)` [Function]

Return the number of bytes allocated since the last garbage collection.

`gc-incremental-traversal-threshold` [Variable]

Number of elements processed in one cycle of incremental traversal.

Statistics

Statistics to examine and debug memory manager issues:

`(gc-stats)` [Function]

Return statistics about garbage collection cycles in a property list.

`(show-gc-stats)` [Function]

Pretty-print statistics about garbage collection cycles to a buffer by using `(gc-stats)`.

`(lrecord-stats)` [Function]

Return statistics about Lisp object memory usage in a property list.

`(show-lrecord-stats)` [Function]

Pretty-print statistics about Lisp object memory usage to a buffer by using `(lrecord-stats)`.

Appendix B

Detailed Results of Measurements

I made some measurements to document the current state of the work and to collect information that may help research how the new incremental garbage collector can be optimized in future work. I documented the results for startup, benchmark, and regression tests, always in comparison to the non-incremental traditional garbage collector.

Note that collecting and printing the data slows down execution.

- At the beginning and the end of each step of garbage collection (see section 3.3.4), and with every 100th allocation of a Lisp object a record is printed out that contains the following information:
  - time since startup
  - name of current phase
  - current heap memory usage
  - current virtual memory usage as retrieved from `/proc`

  This provides information about the memory usage and where the execution time is spent.
- Each processed Lisp object is counted whenever it is
  - pushed onto the mark stack,
  - removed from the mark stack,
  - re-pushed by the write barrier,
  - traversed by the second traversal of the root set,
  - freed by the garbage collector,
  - freed manually by the client outside the garbage collector, or
  - it was not freed although the client tried to (see section 3.3.18).

  The Lisp object counts are printed at the end of the test run.

The recorded data is used to plot the graphs shown in chapter 4. The results mentioned in the previous chapters are calculated based on the collected data. In the following, I present the measurements in detail.

B.1 Startup

Starting an Editor should be fast. Do the modifications to the memory manager alter startup performance?

Command:

```
xemacs -vanilla -kill
```

Results

Memory Usage

The memory-usage statistics show information about the average and maximum heap and virtual-memory sizes.

Memory Usage	Heap		Virtual Memory	
	average	maximum	average	maximum
non-incremental	4.94 MB	6.02 MB	33.41 MB	79.16 MB
incremental	4.94 MB	6.49 MB	33.56 MB	79.16 MB
+/−	±0%	+8%	±0%	±0%

Program Execution Performance

The program-execution-performance statistics show information about the total program runtime and the respective shares of client and garbage collection execution time.

Execution Time	Total Execution Times		
	Total	Client (%)	GC (%)
non-incremental	9.29 s	7.06 s (76%)	2.24 s (24%)
incremental	8.99 s	7.06 s (78%)	1.93 s (22%)
+/−	−3%	±0%	−14%

Garbage Collection Phases Execution Performance

The phases execution performance statistics show information about the respective shares of mark and sweep phase execution times.

GC Phase Times	GC	Phases Execution Times		
	Total	Mark (%)	Sweep (%)	Misc (%)
non-incremental	2.24 s	1.88 s (84%)	0.18 s (8%)	0.18 s (8%)
incremental	1.93 s	1.63 s (84%)	0.16 s (8%)	0.15 s (8%)
+/−	−14%	−13%	−12%	−18%

Garbage Collection Times

The garbage collection times statistics show information about how many full garbage collections and how many garbage collection cycles occur. With this information, the time per cycle can be calculated. Time per cycle is the average client pause time.

GC Times	Number of GCs	Number of Cycles	Time in GC	Time per GC	Time per Cycle
non-incremental	11	11	2.24 s	203 ms	203 ms
incremental	9	23	1.93 s	215 ms	84 ms
+/−	−18%	+109%	−14%	+6%	−59%

Lisp Object Statistics

The Lisp object statistics show how many objects the mark algorithm traverses, how many objects have to be re-pushed that have been modified by the client during a garbage collection, and how many objects the collector re-examines during completion of the traversal with re-pushing the root set. The freeing statistics show how many objects the garbage collector actually frees and how many temporary objects the client explicitly frees outside the garbage collector. The number of Lisp objects not manually freed denotes how many objects the client wanted to free but was not allowed to because the write barrier was running.

Traversal Statistics	Number of Lisp objects traversed	re-pushed by write barrier	re-pushed by root set
non-incremental			
total	2,323,369	-	-
per GC	211,215	-	-
per Cycle	211,215	-	-
incremental			
total	1,926,876	48,346	299
per GC	214,097	5,372	33
per Cycle	83,777	2,102	13

Freeing Statistics	Number of Lisp objects freed in GC	freed manually	not manually freed
non-incremental			
total	150,348	520,902	-
per GC	13,688	-	-
incremental			
total	218,640	454,397	66,394
per GC	24,293	-	-

B.2 XEmacs Benchmark Suite

The XEmacs Benchmark Suite is described in detail in section 4.3.1. It gives a general overview about the overall performance of XEmacs.

Command:

```
xemacs -vanilla -l /home/crestani/src/xemacs/bench/bench.el \
       -eval "(progn (bench 1) (gc-full))" -kill
```

Results

Memory Usage

The memory-usage statistics show information about the average and maximum heap and virtual-memory sizes.

Memory Usage	Heap		Virtual Memory	
	average	maximum	average	maximum
non-incremental	7.68 MB	17.34 MB	54.43 MB	79.41 MB
incremental	11.64 MB	63.74 MB	44.37 MB	97.71 MB
+/−	+52%	+268%	−18%	+23%

Program Execution Performance

The program-execution-performance statistics show information about the total program runtime and the respective shares of client and garbage collection execution time.

Execution Time	Total Execution Times		
	Total	Client (%)	GC (%)
non-incremental	82.39 s	62.63 s (76%)	19.76 s (24%)
incremental	78.99 s	63.37 s (80%)	15.62 s (20%)
+/−	−4%	+1%	−21%

Garbage Collection Phases Execution Performance

The phases execution performance statistics show information about the respective shares of mark and sweep phases execution times.

GC Phase Times	GC	Phases Execution Times		
	Total	Mark (%)	Sweep (%)	Misc (%)
non-incremental	19.76 s	15.95 s (81%)	2.15 s (11%)	1.66 s (8%)
incremental	15.62 s	12.23 s (78%)	2.06 s (13%)	1.33 s (9%)
+/−	−21%	−23%	−4%	−20%

Garbage Collection Times

The garbage collection times statistics show information about how many full garbage collections and how many garbage collection cycles occur. With this information, the time per cycle can be calculated. Time per cycle is the average client pause time.

GC Times	Number of GCs	Number of Cycles	Time in GC	Time per GC	Time per Cycle
non-incremental	61	61	19.76 s	324 ms	324 ms
incremental	43	132	15.62 s	363 ms	118 ms
+/−	−30%	+116%	−21%	+12%	−63%

Lisp Object Statistics

The Lisp object statistics show how many objects the mark algorithm traverses, how many objects have to be re-pushed that have been modified by the client during a garbage collection, and how many objects the collector re-examines during completion of the traversal with re-pushing the root set. The freeing statistics show how many objects the garbage collector actually frees and how many temporary objects the client explicitly frees outside the garbage collector. The number of Lisp objects not manually freed denotes how many objects the client wanted to free but was not allowed to because the write barrier was running.

Traversal Statistics	Number of Lisp objects traversed	re-pushed by write barrier	re-pushed by root set
non-incremental			
total	20,545,307	-	-
per GC	336,808	-	-
per Cycle	336,808	-	-
incremental			
total	15,622,685	161,369	3,770,117
per GC	363,318	3,753	87,677
per Cycle	118,358	1,222	28,561

Freeing Statistics	Number of Lisp objects freed in GC	freed manually	not manually freed
non-incremental			
total	7,975,725	9,945,429	-
per GC	163,040	-	-
incremental			
total	8,239,090	9,649,673	250,741
per GC	225,464	-	-

B.3 Regression Tests

XEmacs provides *regression tests* to allow developers to check if their changes lead to any unexpected regressions in other parts of XEmacs [WTB+]. These tests provide a log of passed and failed tests, which allow the developer to investigate the source of the error and fix the bug. Each important feature of XEmacs has its own test cases. Since these tests push XEmacs to its limits, they provide interesting results.

Command:

```
xemacs -vanilla -batch -l ../tests/automated/test-harness.el \
       -f batch-test-emacs ../tests/automated
```

Results

Memory Usage

The memory-usage statistics show information about the average and maximum heap and virtual-memory sizes.

Memory Usage	Heap		Virtual Memory	
	average	maximum	average	maximum
non-incremental	8.02 MB	156.41 MB	44.50 MB	241.71 MB
incremental	8.24 MB	158.72 MB	44.35 MB	241.71 MB
+/−	+3%	+1%	±0%	±0%

Program Execution Performance

The program-execution-performance statistics show information about the total program runtime and the respective shares of client and garbage collection execution time.

Execution Time	Total Execution Times		
	Total	Client (%)	GC (%)
non-incremental	81.48 s	33.24 s (41%)	48.24 s (59%)
incremental	79.22 s	34.07 s (43%)	45.15 s (57%)
+/−	−3%	+2%	−6%

Garbage Collection Phases Execution Performance

The phases execution performance statistics show information about the respective shares of mark and sweep phases execution times.

GC Phase Times	GC	Phases Execution Times		
	Total	Mark (%)	Sweep (%)	Misc (%)
non-incremental	48.24 s	35.14 s (73%)	6.05 s (13%)	7.05 s (15%)
incremental	45.15 s	32.48 s (72%)	6.12 s (14%)	6.55 s (15%)
+/−	−6%	−8%	+1%	−7%

Garbage Collection Times

The garbage collection times statistics show information about how many full garbage collections and how many garbage collection cycles occur. With this information, the time per cycle can be calculated. Time per cycle is the average client pause time.

GC Times	Number of		Time	Time per	
	GCs	Cycles	in GC	GC	Cycle
non-incremental	163	163	48.24 s	296 ms	296 ms
incremental	143	298	45.15 s	316 ms	152 ms
+/−	−12%	+83%	−6%	+7%	−49%

Lisp Object Statistics

The Lisp object statistics show how many objects the mark algorithm traverses, how many objects have to be re-pushed that have been modified by the client during a garbage collection, and how many objects the collector re-examines during completion of the traversal with re-pushing the root set. The freeing statistics show how many objects the garbage collector actually frees and how many temporary objects the client explicitly frees outside the garbage collector. The number of Lisp objects not manually freed denotes how many objects the client wanted to free but was not allowed to because the write barrier was running.

Traversal Statistics	Number of Lisp objects		
	traversed	re-pushed by write barrier	re-pushed by root set
non-incremental			
total	43,223,872	-	-
per GC	265,177	-	-
per Cycle	265,177	-	-
incremental			
total	38,727,423	387,198	300,695
per GC	270,821	2,708	2,103
per Cycle	129,958	1,299	1,009

Freeing Statistics	Number of Lisp objects		
	freed in GC	freed manually	not manually freed
non-incremental			
total	6,689,633	776,734	-
per GC	41,041	-	-
incremental			
total	6,786,902	681,502	776,632
per GC	47,461	-	-

Bibliography

[App] Apple. *Mac OS X Homepage*. `http://www.apple.com/macosx` [Online; accessed 31-Jul-2005].

[BGH+88] David L. Black, David B. Golub, Karl Hauth, Avadis Tevanian, and Richard Sanzi. The mach exception handling facility. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging*, pages 45–56, New York, NY, USA, 1988. ACM Press.

[Boe] Hans-J. Boehm. *A garbage collector for C and C++*. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/index.html` [Online; accessed 31-Jul-2005].

[Cre04] Marcus Crestani. *Ein neuer Speicher-Allokator für XEmacs*. Studienarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2004.

[Cyg] *Cygwin Homepage*. `http://www.cygwin.com` [Online; accessed 31-Jul-2005].

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Sholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. In *Communications of the ACM*, pages 21(11):966–975, 1978.

[FSF01] *The GNU C Library Reference Manual*. Free Software Foundation, 2001. `http://www.gnu.org/software/libc/manual` [Online; accessed 31-Jul-2005].

[Hay97] Barry Hayes. Ephemerons: A new finalization mechanism. In *Object-Oriented Programming, Systems, Languages and Applications*, pages 176–183, 1997.

[IG04] IEEE and Open Group. *IEEE Std 1003.1*. The IEEE and The Open Group, 2004.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Ltd., 1996.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 2 edition, 1988.

[LAX] *Archived XEmacs Mailing Lists*. `http://list-archive.xemacs.org` [Online; accessed 31-Jul-2005].

[man] *Manual pages for several UNIX/Linux system calls*. Part of all UNIX/Linux distributions.

[PLT] PLT. *PLT MzScheme Homepage*. `http://www.plt-scheme.org/software/mzscheme` [Online; accessed 31-Jul-2005].

[Pre00a] Microsoft Press. *AddVectoredExceptionHandler*. Microsoft Developer Network (MSDN) Library, 2000. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/debug/base/addvectoredexceptionhandler.asp` [Online; accessed 31-Jul-2005].

[Pre00b] Microsoft Press. *Exception Handling Mechanisms*. Microsoft Developer Network (MSDN) Library, 2000. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccelng/htm/key_s-z_3.asp` [Online; accessed 31-Jul-2005].

[Pre00c] Microsoft Press. *VirtualProtect*. Microsoft Developer Network (MSDN) Library, 2000. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/virtualprotect.asp` [Online; accessed 31-Jul-2005].

[Rei01] Richard Reingruber. *Alternative Speicherverwaltung für XEmacs*. Diplomarbeit, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, 2001.

[RRSF00] Gustavo Rodriguez-Rivera, Mike Spertus, and Charles Filterman. *Conservative Garbage Collection for General Memory Allocators*. ISMM2000 International Symposium on Memory Management, 2000.

[SGG03] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 6th edition, 2003.

[SGG04] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 7th edition, 2004. Appendix B: The Mach System. Available Online: `http://www3.interscience.wiley.com:8100/legacy/college/silberschatz/0471694665/appendices/appb.pdf` [Online; accessed 31-Jul-2005].

[Ste75] Guy L. Steele. Multiprocessing compactifying garbage collection. In *Communications of the ACM*, pages 18(9):495–508, 1975.

[SW] Richard Stallman and Ben Wing. *XEmacs User's Manual*. Part of the XEmacs distribution.

[Wik05] Wikipedia. *POSIX*. Wikipedia, the free encyclopedia, 2005. `http://en.wikipedia.org/wiki/POSIX` [Online; accessed 31-Jul-2005].

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42, St. Malo, France, 1992. Springer-Verlag Lecture Notes in Computer Science no. 637.

[WLLS] Ben Wing, Bil Lewis, Dan LaLiberte, and Richard Stallman. *XEmacs Lisp Reference Manual*. Part of the XEmacs distribution.

[WTB+] Ben Wing, Stephen Turnbull, Martin Buchholz, Hrvoje Niksic, Matthias Neubauer, Olivier Galibert, and Andy Piper. *XEmacs Internals Manual*. Part of the XEmacs distribution.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. In *Journal of Systems and Software*, pages 11:181–198, 1990.

Index