TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

JUSSI LAHDENNIEMI

PARALLEL GARBAGE COLLECTING MEMORY MANAGER FOR C++

Master of Science Thesis

Topic accepted in the department council meeting 7.5.2003

Examiners: Prof. Ilkka Haikala (TUT)
Prof. Kai Koskimies (TUT)

Preface

I have constructed my master's thesis as an employee at Ionific Ltd., Tampere. Originally, the work was started as a part of a bigger software development project, but the resulting application can also be used in other contexts. I want to thank my colleagues at Ionific as well as professors Ilkka Haikala, Kai Koskimies and Antti Valmari at the Tampere University of Technology for their ideas and feedback. Special thanks go to my wife, Eeva, for moral support.

Tampere, 16.5.2003

Jussi Lahdenniemi
Riekontie 5
FI-39160 Julkujärvi
FINLAND

tel. +358 50 591 1448
fax. +358 (3) 318 6100
e-mail jussi.lahdenniemi@ionific.com

Contents

Preface . . . ii
Abstract . . . v
Tiivistelmä . . . vi
1. Introduction . . . 1
2. Garbage Collection Techniques . . . 4
2.1. Garbage collection process . . . 4
2.2. Copying versus mark-and-sweep collectors . . . 5
2.3. Programming language support . . . 6
2.4. Partitioning the collection . . . 7
2.5. Optimising the collector . . . 8
2.6. Additional features . . . 9
3. Basic Implementation . . . 10
3.1. Design choices . . . 10
3.2. Heap organisation . . . 11
3.3. Function interface . . . 14
3.4. Memory allocation . . . 14
3.5. Garbage collection . . . 17
4. Concurrency and Scalability . . . 22
4.1. Atomic operations . . . 22
4.2. Concurrent allocation . . . 23
4.3. Concurrent allocation of large objects . . . 25
4.4. Concurrent collection . . . 28
4.5. Basic performance optimisations . . . 35
5. Evaluation . . . 38
5.1. Concepts . . . 38
5.2. Case study 1: Multithreaded memory allocator . . . 39
5.3. Case study 2: GNU AWK . . . 41

5.4. Case study 3: Boolean Expression Reduction . . . 42

5.5. Summary . . . 44

6. Related Work . . . 45

6.1. Garbage collection and C++ . . . 45

6.2. Parallel collection . . . 46

7. Conclusions . . . 47

References . . . 49

Appendix A. Interface header . . . 51

TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Software Systems

LAHDENNIEMI, JUSSI: Parallel Garbage Collecting Memory Manager for C++

Master of Science Thesis, 48 pages, 4 appendix pages.

Examiners: Prof. Ilkka Haikala and Prof. Kai Koskimies

May 2003

Keywords: parallel computing, garbage collection, automatic memory management

In the design of a programming language one of the important decisions is the language's view to dynamic memory management. In this respect, languages divide into two main groups. Languages with manual memory management forward the memory management responsibility to the programmer creating applications with the language, making the language easier to implement and more compact. Other languages automatically manage memory, requiring no effort from the applications programmer. While the run-time libraries of such languages are larger and they use somewhat more memory, automatic memory management has been shown to bring down development times and reduce memory related bugs in programs.

In this thesis work, a garbage collecting memory manager was designed and implemented for C++ that normally only offers manual memory management functionality. Special attention was devoted to the parallel computing aspects of the software, improving its scalability in multi-processor systems. In the thesis, algorithms are presented that enable lock-free implementation of the data structures necessary for the manager's functionality. Some initial benchmarking results based on the first implementation are also given.

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan osasto

Ohjelmistotekniikka

LAHDENNIEMI, JUSSI: Rinnakkainen roskia keräävä muistinhallintajärjestelmä C++:lle

Diplomityö, 48 sivua, 4 liitesivua.

Tarjastajat: Prof. Ilkka Haikala ja Prof. Kai Koskimies

Toukokuu 2003

Avainsanat: rinnakkainen laskenta, roskienkeruu, automaattinen muistinhallinta

Ohjelmointikieltä suunniteltaessa yksi tärkeistä päätöksistä on muistinhallintastrategian valinta. Manuaalisen muistinhallinnan sisältävät kielet siirtävät hallintavastuun kieltä käyttävälle ohjelmoijalle, mikä tekee kielestä kompaktimman ja helpomman toteuttaa. Toisaalta kielet, joiden määrittely sisältää automaattisen muistinhallinnan, päästävät ohjelmoijan helpommalla, kun kieli itsessään huolehtii muistin vapauttamisesta. Automaattinen muistinhallinta lyhentää ohjelmistojen kehitysaikoja ja vähentää muistinkäsittelyyn liittyviä ohjelmointivirheitä.

Tässä diplomityössä suunniteltiin ja toteutettiin roskia keräävä muistinhallintajärjestelmä C++:aan, jossa on normaalisti käytössä vain manuaalinen muistinhallinta. Erityistä huomiota kiinnitettiin ohjelmiston rinnakkaiseen toimintaan, jotta skaalautuvuus olisi moniprosessorikoneissa mahdollisimman hyvä. Diplomityössä esitellään tähän tarkoitukseen kehitetyt lukituksettomat algoritmit tarvittaville tietorakenteille. Työ sisältää myös toteutetun muistinhallinnan ensimmäisen version alustavia nopeustestituloksia.

1. Introduction

In the design of a programming language one of the important decisions is the language's view to dynamic memory management. This feature is essential to many real-world data processing applications. The design and implementation of memory management functionality is important for the success of the language.

In languages with explicit memory management the programmer is responsible for freeing all allocated memory later in the program when it is no longer needed. With automatic memory management, the programmer can keep allocating blocks and using them as necessary without worrying about freeing them. The memory manager tracks all allocated memory and reuses it after it can no longer be reached from the program code. This process is called *garbage collection*, which is often used as a synonym to automatic memory management.

Some programming languages such as Lisp and Smalltalk have traditionally had automated memory management. More modern examples of this design are Java and Microsoft's C#. Automatic memory management relieves the programmer from the burden of looking after the blocks or memory allocated by the program. On the other hand, automatic memory management is much more difficult to implement than the explicit counterpart. For example, the popular programming languages C and C++ have chosen the way of explicit memory management in accordance to their design principles of giving the user a total control of the program execution.

Automatic memory management is often associated with interpreted languages, slowness and inefficiency. The average programmer's ad-hoc feeling is that he can manage the memory much better than the computer can. While this assumption may sometimes be correct, explicit memory management brings other problems along: Annoying memory-related bugs

are common and often hard to locate and eliminate. It is easy to forget to free the unneeded memory blocks, which leads to memory leaks. In programs running for short periods only, these leaks may not cause harm but in server systems where programs are left running for weeks or months even small leaks can accumulate and bring down the program.

While explicit memory management may gain something in performance and memory use, automatic memory management has been shown to reduce development times [15]. Writing code in a system equipped with an automatic memory manager is easier and less prone to errors. This is because the memory manager takes care of tedious bookkeeping tasks that in an explicitly managed heap are left to the application developer. Also, while automatic memory management is usually somewhat slower than explicit one, the difference is surprisingly small [23].

The biggest drawback in automated memory management schemes, compared to explicit memory management, is the fact that the total memory heap size is larger [23]. This may be a problem in some environments. However, the heap is only larger by a constant factor, so unless the memory is very limited and virtual memory is not available, the size of the heap is not a limiting issue.

In this thesis, the implementation of a garbage collecting memory manager for C++ is described. The memory manager is designed to be part of a platform for building the server-side applications of client-server software. However, it does not require support from the rest of the platform, and can thus be used as-is in other applications.

As the C++ language does not natively support automated memory management, implementing one is a relatively painful operation. Also, the bulk of research in the field of automated memory management is directed to other efforts. There are, however, at least two competitive implementations available for public use [4, 5, 6, 9]. Before starting the design of the memory manager discussed in this thesis, these two managers were assessed, but to some extent found lacking features needed in the target system.

The first focus point of the target implementation is ease of use. The memory manager is intended to be included in a software development kit distributed to software developers, and it is important that the developers can easily adopt the conventions of the product and thus do not disregard it as too hard or complex. The second focus point of the implementation is performance. The memory manager must be efficient enough to guarantee sufficiently fast response times from the server applications. As server machines with several processors

are common, the implementation must also scale well: The processors must not block each other's access to memory resources. This requires special lock-free data structures and algorithms that were devised for the work and are presented in this thesis.

Chapter 2 gives overall information on the various garbage collection techniques. The design choices of the target implementation are accounted for and compared with other possible methods. Chapter 3 describes the general structure of the basic implementation of the memory manager as well as data structures needed for its operation, while Chapter 4 focuses on the scalability and multi-processor properties of the manager.

Chapter 5 lists some basic performance results from the collector. In Chapter 6, the work is compared to other similar solutions. Chapter 7 summarizes the results of the work and takes a glance at the future prospects and possible improvements of the memory manager.

2. Garbage Collection Techniques

2.1. Garbage collection process

As the details of the various techniques differ, the broad idea of general-purpose automatic memory management stays the same. As with all memory management, the process is divided into two main phases: Allocation and freeing of memory. In manual memory management systems, the programmer originates both, while in garbage-collected environments the system is responsible for freeing the memory.

Memory allocation in garbage-collected environment is straightforward and does not differ much from manually managed environments. Objects are allocated from free locations in the heap. Usually some internal bookkeeping is also needed so that the collector will later be able to determine the locations and sizes of allocated memory blocks.

Garbage collection can be triggered by several causes. The most obvious approach is to collect garbage when the heap's size has grown over a predefined limit from the situation after the previous collection. Another approach is to collect garbage at predefined time intervals; collection can also be timed to happen when the system does not have anything else to do.

The garbage collector starts a collection cycle by determining the root set of data from which references to heap objects can originate. These roots include the thread stacks as well as modules' static data areas and memory allocated by other means than the collector. Finding this root set depends greatly on the environment: In case the programming language and environment fully support garbage collection, it can be trivial – on the other hand, an add-on garbage collector may need to scan lots of data that has nothing to do with the heap.

The next step for the collector is to trace all references from the root set to objects in the memory manager's managed heap. These objects are now known to be referable by the program code and must thus be preserved. They are scanned in turn to uncover any references to other heap objects, and the process continues until there are no references to non-scanned objects.

The garbage collector now knows that all other objects present in the heap are garbage, as the program code has no legal way of addressing them. These objects can be freed and reused the next time memory needs to be allocated. After this, the garbage collection cycle is finished and the control returns to the application.

Naturally, there is more than one way to accomplish this process. All approaches view the problem from a different angle, and the correct approach depends on the tools and needs of the implementer. [19]

2.2. Copying versus mark-and-sweep collectors

The first major division of collectors is made between copying and mark-and-sweep ones. Some hybrid solutions have also been suggested, but usually the implementation clearly falls into either of these two categories.

Copying collectors are the simpler kind. Basically, they divide the heap space into two hemi spaces. At any given time, only one of these hemi spaces is used for new object allocation. When garbage collection starts, objects are copied from the allocation space (usually called *from-space*) to the other, initially empty one (*to-space*), as they are determined to be reachable by the application. In the process, pointers to the objects are also changed to point to the new hemisphere. After all reachable objects have been copied, the roles of these hemi spaces are switched, and collection finishes.

For a copying collector, memory allocation is trivial: The top of the heap is stored in a variable, and every time an allocation happens, the variable's value is increased by the allocation's size and the original value is returned.

Mark-and-sweep collectors perform the collection in two phases. At the beginning of a collection cycle all memory allocations are marked as *not seen*. As the collection progresses, the referenced allocation blocks are marked as *seen*, until all references have been traced. The mark phase is now complete, and the sweep phase sweeps away all allocations that have not been marked as *seen*. These blocks can now be reused.

This technique needs substantially more bookkeeping than copying collection. Allocation is more complicated, as the heap can get fragmented when blocks are freed in non-sequential order. On the other hand, pointers need not be rerouted and memory needs not be copied, as is the case with the copying collector.

Generally, mark-and-sweep collectors are more complex and harder to implement. However, in performance, they are on the same line with copying collectors that have to perform extra work when moving the heap objects around [21]. Thus, the underlying hardware or the language for which the collector is aimed usually determines the appropriate implementation technique. Naturally, application-specific needs must also be considered.

2.3. Programming language support

Being able to rely on the programming language to provide support makes the job of the garbage collector much easier. Languages that have been designed with automatic memory management usually attach type information in the values of variables. Thus, the collector knows a pointer when it sees one, and can be sure that the value really is a pointer to a heap object. For a language supporting automatic memory management it is possible to implement a *precise* garbage collector. After a precise collector has completed a collection run, exactly those objects that can be referenced by the application will be left alive, while unreachable objects are guaranteed to be collected.

Collectors that do not rely on type information provided by the language environment are called *conservative*. They treat each machine word the same way: It Might Be a Pointer. If a machine word's contents interpreted as a pointer refer to an allocated block of memory, that block is treated as being referenced. A purely conservative collector cannot be a copying collector, because it cannot adjust variables' contents – adjusting a variable that, after all, is not a pointer is fatal to the application. Also, a conservative collector can never be sure it has really collected all garbage objects, because a random, unused machine word can be interpreted as a live pointer to a heap object.

In this work, the target language is C++. As C++ does not support generic runtime type information, the collector must be conservative. As a matter of fact, John F. Bartlett has introduced a mostly copying collector for C and C++ [5]. However, this approach relies on the user to provide the type information. Thus, a ready-made application cannot just be recompiled to get it work with this kind of a mostly copying collector. A conservative

mark-sweep collector is somewhat more complicated to implement, but it is the only choice if the aim is set to a general-purpose collector.

2.4. Partitioning the collection

If the garbage collection is performed in one piece, execution of the program stops abruptly, usually during a memory allocation function call. In a program simply processing loads of data this is not a problem, but in case of an application with real-time requirements or even a user interface that should not stop responding randomly while the garbage is collected, a more sophisticated method is required. There are two main ways of accomplishing this.

An *incremental* collector performs some collection work every time memory is allocated. This way, collection advances abreast allocation. When the collection is ready, new allocations have been made, and a new collection cycle can be initiated. The collection work can be divided proportionally to the size of each allocation. This way, memory allocation takes time linearly proportional to the size of the heap, and a new collection starts immediately after the previous one completes. This way the overhead from the collector stays exactly the same over time and estimating the performance hit is easy. This technique can even be used to implement memory management for hard real-time systems [1, 2].

In *parallel* collection the memory manager spawns a new thread for the use of the collector. While application threads allocate memory, the collector's thread works in the background collecting garbage. The collector thread's priority can be adjusted to match the current memory situation: If the collector falls behind and the heap seems to grow too fast, the collector thread is prioritised for more CPU time. Also, if the application's CPU use is periodical, the idle time is not wasted as the collector thread can utilize the time for collection work. As early as 1978, Dijkstra et al. published a paper on parallel garbage collection [10].

Independent of the method, partitioning the collection requires special techniques to handle situations where the mutators alter contents of memory already scanned by the collector. If a mutator copied a pointer from a non-scanned memory block into a scanned block, the collector would never see the pointer and might erroneously regard the referenced block as garbage.

Implementers fortunate enough to build memory managers for platforms where not alone the programming language but the hardware supports garbage collection, solve the problem with read and write barriers [22]. These barriers generate hardware interrupts when the application

attempts to read or write beyond the address where the barrier currently lies. This way the collector can react to the mutation, for example by rescanning the touched memory position. Simple barriers work best with copying collection, where scanned and non-scanned objects are easily distinguished by their location in the memory.

On platforms where barriers are nonexistent they are often emulated with normal memory protection functionality. The scanned pages are simply marked as read-only, in case of write barrier, or totally inaccessible for a read barrier. The hardware generates an interrupt for the access violation and the collector knows that the page has been touched. This method results in somewhat more work than with the barriers, because the collector must keep track of individual pages' protection status. Also, operating systems may not be optimised for this kind of interrupt processing especially in the user mode, which makes the operations slow.

2.5. Optimising the collector

There are usually two kinds of memory blocks allocated: Those that live for a very short time and those that stay alive long. An example of the first category is a temporary buffer allocated for a function; a context object for a session in a client-server application belongs to the latter category. Thus, it is often sensible to quickly try and sweep away the young objects that have a high probability of being garbage, and not check the older objects all the time: If they have stayed alive for 42 collections, the probability of them dying on 43rd is not high. [12]

The two groups could be divided even further. Youngest objects might be processed on every collection; those that have survived five collections would be handled on every third collection, while older objects with an age over twenty collections are only checked every tenth time and ancient creatures with their age exceeding a hundred collections are never scanned. These groups are called generations. Generational collection is a very common technique of speeding up garbage collectors. [3]

Another important optimisation technique concerning conservative collectors is *blacklisting*. When the collector scans data roots and allocated blocks, it may encounter words that do not point to any currently allocated blocks (i.e. are not pointers), but that do point to the memory manager's heap area. Should the heap later grow, these non-pointers would then point to the active heap effectively preventing collection of a suitably allocated block. Although this is not an error – for a conservative collector, it is basically never an error not to reclaim a block of memory – it is undesired, since unused memory is not reclaimed.

As such, when the collector encounters a word that will potentially be dangerous later, it can blacklist the page of its heap area that the word references. Blacklisted pages are never used for normal allocation, although they can be used for the manager's internal data structures that are not collected normally.

2.6. Additional features

User of a garbage collecting memory manager sometimes needs additional control over the collector. It is often useful to offer the programmer the possibility to decide when a garbage collection cycle is to be initiated: Collections can be performed when the application is known to sit idle for some time. On the other hand, some applications may need a way to prohibit collections for the duration of a specifically time-critical piece of code.

References to objects coming from outside the application's scope, such as file handles given by the operating system, pose a problem to the application developer using an automated memory management system. As the memory manager frees allocated objects, the programmer has no way of automatically disposing of handles embedded in heap objects. *Finalisation* functionality solves this problem. Finalisation functions can be associated with allocated objects; they are called when the allocated object is collected.

When targeting C++ programmers, it is important to stress the difference between destructors and finalisation. The finalisation function registered with an object is not necessarily called immediately after the last reference to the object disappears; actually, it may not even get called during the first year from the disappearance of the last reference. As such, finalisation must not contain critical cleanup code, but merely code protecting system resources from getting wasted in large numbers.

3. Basic Implementation

3.1. Design choices

The goal of the work described in this thesis is an easy-to-use automatic memory manager for C++. As the language does not provide type information or any other support for automatic memory management, a conservative garbage collector is the only viable choice. Conservatism further limits the collection technique into mark-and-sweep collection.

In the first phase, the implementation is not specially optimised. A minimal set of functionality is implemented and tested. Moreover, the code is designed so that adding functionality and features and optimising the collection will be as easy as possible.

Even the first implementation must support threads. Although a very naïve approach – use of a single mutual exclusion object – is chosen, the design permits more sophisticated and scalable methods. Parallelism will be added step-by-step after the first version works.

The basic implementation does not depend much on the underlying hardware or operating system. Mutual exclusion objects needed for multi-thread synchronisation exist in every operating system. The process of finding data roots is a bit more complicated; so is enumerating, pausing and resuming threads in the process. All this can be accomplished in the three main operating systems this work targets to: Microsoft Windows 2000, GNU/Linux and Hewlett-Packard's HP-UX – although, as noted later, the HP-UX places some obstacles on the way when moving on to parallel collection. Operating system dependent code is wrapped in separate header and source files in such a manner that supporting new operating systems is straightforward. The first implementation will be written for Windows 2000.

Finalisation functionality is not included in the first version. However, another feature not discussed in Section 2 is implemented. Allocated objects can be locked and unlocked. Locked objects are not collected even if there are no references to them.

Locking is necessary when the objects are passed to outside functions, especially operating system level functions. When this happens, references to the allocated blocks may get lost from the collector, but return later when the operating system returns the object to the application. An example of this is an asynchronous I/O transaction: The application allocates a buffer for the I/O and calls a system function that initiates the operation and returns immediately. When the operation completes, the operating system signals the completion to the application and gives the buffer pointer back to the application. This way, the application needs not store the buffer pointer when the operation lasts. However, as the garbage collector cannot reach the operating system's kernel data structures, it sees no references and frees the object.

Yet another extra functionality is added to the design. Allocated blocks can be tagged to contain no pointers. This is often useful when large buffers are allocated for I/O or data storage: Being able to ignore a two-megabyte buffered JPEG image when scanning the heap speeds up collection considerably.

3.2. Heap organisation

When initialised, the memory manager reserves a large continuous region of the application's virtual address space. The size of this region sets the limits for the memory available to the application, while growing the size does not dramatically raise the memory requirements for the collector. Thus, it is in 32-bit systems often sensible to reserve a region as large as the system permits. In 64-bit systems virtual address space is even cheaper and a sufficiently large region can easily be allocated.

The heap is divided into *macro blocks* of a constant size. The size of the macro blocks must be a multiple of the underlying hardware's physical page size. Other than that, deciding the size of the macro blocks is more or less arbitrary. Larger macro blocks are good when allocations are often of the same size, while smaller macro blocks save space when allocation sizes vary. The basic implementation opts for eight-kilobyte macro blocks.

All allocation blocks within a macro block are equally sized. Each macro block begins with a header containing the size of the macro block's allocations. Additionally, each allocation block starts with a header structured as shown in Table 1.

Table 1. Allocation block header structure.

Member	Description
next	Doubly linked list's *next* pointer.
prev	Doubly linked list's *previous* pointer.
type	The block's type. For example, this field flags blocks marked as containing no pointers.
colour	The colour of the block. As suggested by Dijkstra [10], there are four colours: Black, grey, white and green.

Allocations in the heap are divided into two groups: Small and large allocations. The macro block size defines the threshold: All allocations fitting into a macro block are small, while allocations that need two or more macro blocks are considered large. Naturally, large allocation's macro blocks other than the first one cannot contain macro block headers – they would be inside the allocation's area. The manager keeps macro block header presence flags in a single array with one bit for every macro block in the heap area: If the bit is set, the corresponding macro block has a header. Those macro blocks that do not have their bit set belong to a large allocation beginning from an earlier macro block.

The memory manager has a single global data structure holding the settings and state of the manager. The contents of the global data structure are shown in Table 2.

Table 2. The global data structure of the memory manager.

Member	Description
heapStart, heapEnd	These two members contain information about the heap's location in the virtual address space. They point to the start and end, respectively, of the address space reserved for the heap.
heapTop, heapSize	As the heap grows, *heapTop* tracks the highest address currently committed to the heap, while *heapSize* holds the total size of the virtual address space committed to the heap.

Member (cont.)	Description
currentHeapSize, lastGcHeapSize	A new garbage collection cycle is initiated when the heap's allocations' total size exceeds the heap size after the last collection cycle by a predetermined amount. To facilitate this operation, these two member variables hold the said heap sizes.
black, grey, white, green, lock	All allocation blocks in the heap belong to one of the doubly linked lists whose heads are held in these variables. The list always matches the allocation block's current colour. There is only one *black*, *grey*, *white* and *lock* list, but several *green* lists, one for each supported small allocation size and one for large allocations.
colour.black, colour.grey, colour.white, colour.green	For optimisation purposes, the numeric values of the different allocation block colours vary. The values currently in effect are stored in these variables.
threads	Information of the currently running threads is needed so that the collector can correctly determine the data root set. Each thread's stack and register contents belong to the root set. Thread information is stored in this table.
mutex	The collector has to be thread safe, so all memory management functions are protected with a mutual exclusion object. The actual type of this object is operating system dependent.
sizeSizeTable, sizeIdxTable	Each allocation being a multiple of 16 bytes, these tables contain $^{msa}/_{16}$ entries, where *msa* is the maximum size of a small allocation. Each size table entry contains the size rounded upwards to the next quantised size; size index table maps the size to the zero-based ordinal number of the quantised size. See Section 3.4 for further information.
macroBlockHeader	The macro block header lookup table. There is one bit for every macro block in the heap.

3.3. Function interface

The basic implementation offers a simple function interface to the programmer. Functions are provided for allocating memory, locking heap objects and unlocking previously locked objects. Additionally, the different flavours of the global `operator new` are overridden to call the manager's allocation function, and the different versions of `operator delete` are redefined to a null function. This interface allows applications developers to simply add an include statement to the beginning of the project's source files in order to start using the new memory manager. Old source code works as-is, but new code can take advantage of automatic memory management – mainly by forgetting about delete.

When programming in C++, the Standard Template Library (STL) is often used. In STL, memory allocation is handled in special allocator template classes. The interface header defines a generic STL allocator for allocating memory for objects that do not contain pointers. The allocator is specialised for standard C++ data types.

Prototypes for the interface functions are given in Figure 1. The `new` and `delete` operators are actually implemented in-line, the former mapping to an `alloc` call and the latter to an empty function. The STL allocator template, being largish and pretty straightforward, is not included in the listing. However, the whole interface header can be found in Appendix A.

3.4. Memory allocation

3.4.1. General

As described in Section 3.2, memory allocations are divided into two groups: Small and large ones. Allocations that fit into a single macro block are small, while those spanning several macro blocks are large. These two cases are handled somewhat differently and are described in detail in the following two sections.

All memory allocations start with the acquisition of the memory manager's global lock. The lock is held during the whole allocation operation. This prevents other threads from simultaneously trying to allocate memory or start the collection. Obviously, this is not good for multi-processor scalability, but those issues will be addressed in Section 4.

With each allocation unit having a header structure preceding the actual data, the size of the header must be added to the allocation size. This also affects the division between small

```
enum BlockType
{
    Normal, NoPointer
};
void* alloc( size_t size, BlockType type = Normal );
bool lockObject( void* object );
bool unlockObject( void* object );

void* operator new( size_t size );
void* operator new( size_t size, BlockType type );
void* operator new( size_t size, std::nothrow_t const & );
void* operator new( size_t size, std::nothrow_t const &, BlockType type );
void* operator new[ ]( size_t size );
void* operator new[ ]( size_t size, BlockType type );
void* operator new[ ]( size_t size, std::nothrow_t const & );
void* operator new[ ]( size_t size, std::nothrow_t const &, BlockType type );
void operator delete( void * );
void operator delete[ ]( void * );
```

Figure 1. Memory manager's function interface.

and large allocations: The whole small allocation, including the header, must fit into a single macro block.

3.4.2. Small allocations

For small allocations, the total allocation size – that requested by the application plus the size of the header added by the manager – is quantised to one of the precalculated sizes stored in the *sizeSizeTable* of the global data structure. Selecting the set of these sizes is not a trivial matter. If the sizes are spaced too narrowly, allocations sized closely fall into several slots and more allocation blocks are used than necessary. On the other hand, if the spacing is too sparse, allocations are more prone to waste space at their ends. Actually, the optimal set of allocation sizes depends on the application.

The Boehm collector [9] builds its list of supported allocation sizes during application execution. This enables the manager to adapt to each application's specific needs and optimise

memory use for the application's most-used allocation sizes. However, the basic memory manager described in this thesis does not implement this kind of adaptability, using a constant size set defined by Equation 3.1 instead:

$$\left\{ \textsc{Granule} \left\lfloor \frac{\left\lfloor \frac{\textsc{Macro-Block-Size}}{count} \right\rfloor}{\textsc{Granule}} \right\rfloor \quad \text{where} \quad 1 \leq count \leq \frac{\textsc{Macro-Block-Size}}{\textsc{Granule}} \right\} \tag{3.1}$$

For example, with the constant MACRO-BLOCK-SIZE being 8192 and GRANULE being 16 – the values chosen for the first implementation – the set contains 44 items: { 16, 32, 48, 64, 80, ..., 400, 416, 448, 480, ..., 816, 896, 1024, 1168, ..., 2720, 4096, 8192 }. This selection of sizes represents the best set available with the given parameters, but still wastes very much memory especially on very small and very large allocations. For example, a 4097-byte allocation will take 8192 bytes of memory. However, on mid-sized allocations the performance is sufficiently good: Allocations between 16 and 512 bytes only take four per cent extra space on average.

After the quantisation, the correct green object list is selected from the global data structure. That list contains free allocation objects with the correct size. The first object is extracted from the list, unless the list is empty. In case of an empty list, a new macro block is allocated from the heap's virtual address space. The macro block is initialised for that allocation size, and the first allocation block is used for the current allocation. Other blocks are pushed to the green object list for the use of later allocations.

The newly acquired green object is now coloured black. Thus, all blocks allocated after the previous garbage collection are black. After attaching the object to the black objects' list, the allocation unit is ready to be returned to the application. The last operation is releasing the global lock.

3.4.3. Large allocations

Large allocations are quantised to macro block precision. That is, given a macro block size of 8192 bytes, a 10000-byte allocation will take 16384 bytes of heap space. This cannot be avoided, but fortunately large block allocations are relatively rare occurrences compared to small allocations. Also, the relative overhead decreases as the allocations grow.

There is only one green object list for large allocations. The list is always sorted according to the items' positions in the heap and adjacent green objects are merged. This reduces fragmentation of the heap. The list is scanned once, searching for a green object with exactly the size of the new allocation. If such an object is found, the object is removed from the green object list and used for the allocation; if no such object exists, the block with least excess space is chosen and broken into two pieces. The second one, sized correctly for the new allocation, is used for that purpose, while the first part remains in the green object list, with a reduced size. Finally, if the green object list is empty or there are no blocks large enough to accommodate the new allocation request, new macro blocks are claimed from the heap's virtual address space and the allocation is placed there.

The macro block header table reflecting the presence of the macro block headers must be kept up-to-date when the large allocation blocks are fragmented or new ones are allocated. Only the bit for the first macro block of each object is set to one; others are zeroed to signal that the header should be searched earlier in the heap.

After the large allocation object has been chosen and possibly detached from the green object list, it is coloured black and pushed to the black object list like its small colleagues.

3.5. Garbage collection

3.5.1. Mark phase

Garbage collection is triggered by the memory allocation function when the sum of the sizes of allocated objects exceeds the heap size after the previous collection by a predefined ratio.

The first version of the memory manager implements a stop-the-world garbage collection. All other threads are stopped during the execution of the collector and resumed afterwards. This prevents the other threads from modifying the heap's contents while the collector is examining it. It is not important exactly where the threads' execution halts, as long as it does not halt inside the memory manager's own functions – for example, inside another allocation function call. This can be easily ensured with the memory manager's global lock.

Between collections only two colours exist in the heap: Allocated objects are black and free objects are green. After stopping the other threads, the first action for the collector is to invert the white and black colours. Thus, at the beginning of collection only white and green objects exist; white colour denotes objects that have not been seen by the collector while scanning

the heap. The switching of colours is a quick operation, as all colours have their numeric equivalents stored in the global data structure. Switching the blocks' colours is as easy as switching these numbers.

Before the actual marking can begin, the data roots must be found. Data roots are searched in a platform-dependent manner. They consist of the contents of all loaded modules' static and global data areas as well as all threads' stacks and contents of their registers. The data root areas are scanned as they are found. Also the locked objects must be scanned here – they are considered to be a part of the roots.

When a memory area is scanned for heap references, all the correctly aligned pointer-sized entities are checked for a possible reference to a heap object. This operation is relatively fast. The algorithm is given in pseudo code in Figure 2. Most of the checks end right at the first comparison; only machine words whose pointer interpretation points to the area of the currently active heap must be checked further.

The choice to limit the search for only correctly aligned machine words is deliberate. Were the applications programmer to place a pointer in an incorrectly aligned location, the collector would never see it. However, as most hardware architectures suffer from misaligned pointers, and some even refuse to handle them, compilers seldom generate code that automatically creates misaligned pointers: The applications programmer has to create the situation herself. While the memory manager does not expect cooperation from the programmer, it does not except hindrance, either.

Inspection of the mapping algorithm also reveals the rationale of the heap structure described earlier in this chapter. The algorithm may be given any pointer to the heap area: The code must be able to locate the corresponding allocation header structure from the pointer. As macro block headers are always located at the very start of a macro block, it is easy to locate them. All allocations in a macro block being same-sized, the index of the pointed object is easily calculated and the correct allocation object found. For small allocations, all this can be accomplished without iterating any data structures, in constant time.

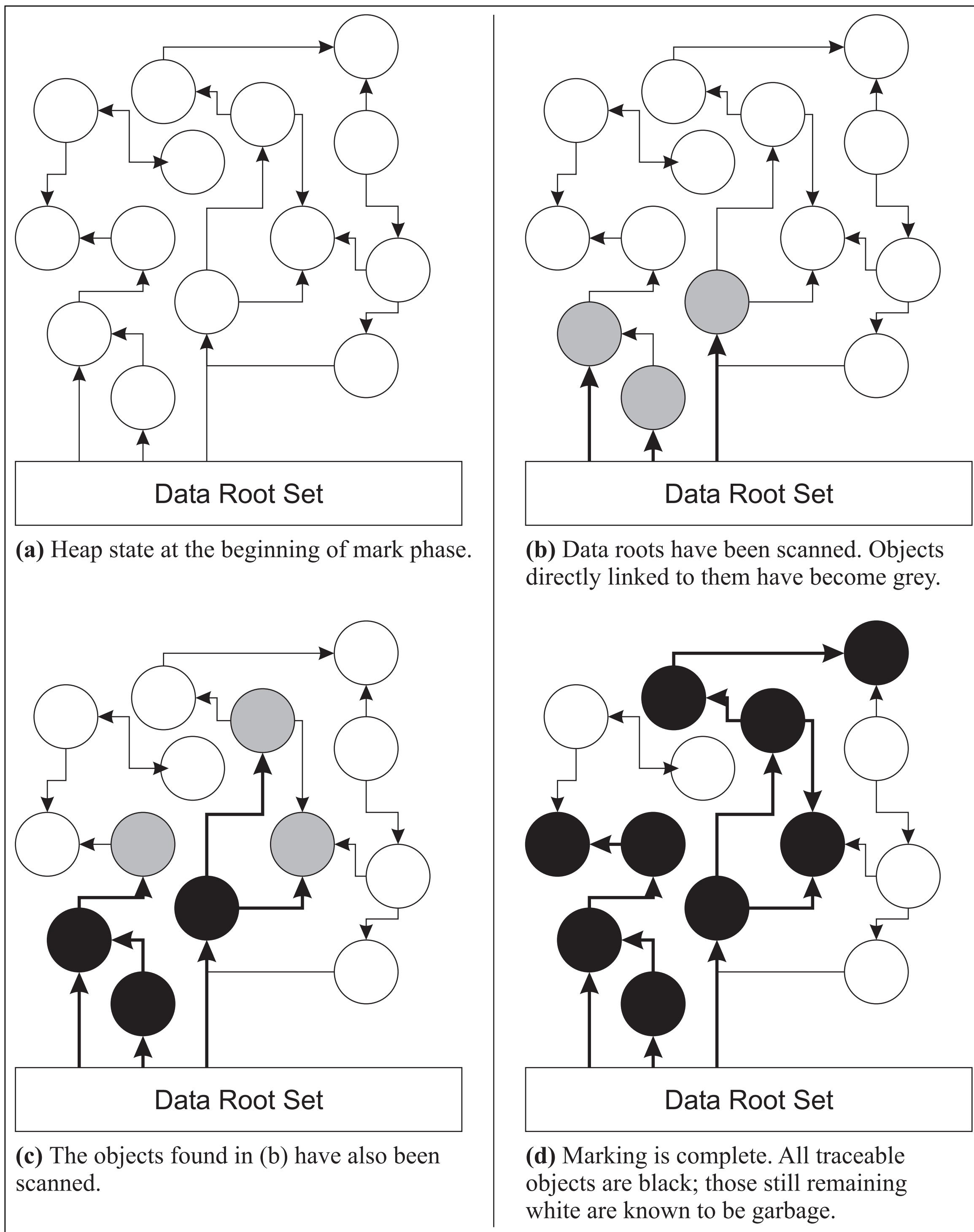
The only case when the algorithm does have to loop occurs when a memory word points into the middle of a large allocation. In this case, the macro block header lookup table must be searched backwards until the first set bit is found that denotes the start of the large allocation.

```
MAP-POINTER(candidate)
1   if candidate < heapStart ∨ candidate ≥ heapEnd
2       then return NIL
3   index ← ⌊(candidate – heapStart) / MACRO-BLOCK-SIZE⌋
4   mbheader ← candidate – (candidate mod MACRO-BLOCK-SIZE )
5   while macroBlockTable[index] = 0
6       do mbheader ← mbheader – MACRO-BLOCK-SIZE
7          index ← index – 1
8   diff ← candidate – mbheader
9   ▷ diff is now the offset from macro block header to candidate.
10  if diff < MACRO-BLOCK-HEADER-SIZE
11      then return NIL
12  diff ← (diff – MACRO-BLOCK-HEADER-SIZE) mod ALLOCATION-SIZE(mbheader)
13  ▷ diff is now the offset from allocation header to candidate.
14  if diff < ALLOCATION-HEADER-SIZE
15      then return NIL
16  alloc ← candidate – diff  ▷ Allocation header.
17  if COLOUR(alloc) = GREEN
18      then return NIL
19  return alloc
```

Figure 2. The function for mapping a pointer into the corresponding allocation block.

When the mapping algorithm returns a result other than *nil,* the returned allocation is assumed to be accessible from the application. If the returned allocation is currently white, it is coloured grey and transferred to the grey object list. Furthermore, white objects marked as containing no pointers can be immediately coloured black and transferred to the black object list. If the object is already grey or black, the collector has already seen it and no further action is needed.

After the data roots have been scanned, the collector can focus on the grey objects. White objects are made grey when the collector first sees them; grey objects are made black after



(a) Heap state at the beginning of mark phase.

(b) Data roots have been scanned. Objects directly linked to them have become grey.

(c) The objects found in (b) have also been scanned.

(d) Marking is complete. All traceable objects are black; those still remaining white are known to be garbage.

Figure 3. An illustration of the garbage collection process.

they have been scanned for references to other heap objects. When grey objects are scanned, new grey objects are pushed to the grey object list, as the collector traces the graph of references through the heap.

Finally, the grey object list is empty. The mark phase is now done: All currently accessible objects are black and inaccessible objects white. The whole garbage collection process is illustrated in Figure 3.

3.5.2. Sweep phase

Compared to the mark phase, sweeping is a breeze. The collector only needs to walk through the white object list now containing all objects that were accessible earlier but are not accessible anymore, colour the objects green and transfer them to the correct green object list according to their size. Only the large object list needs special attention, as consecutive free areas need to be merged. For this purpose, the newly freed objects are sorted according to their location in the heap and then merged with the existing large green object list, combining any successive objects. Sorting a doubly linked list is easily accomplished with merge sort in $O(n \log n)$ time.

4. Concurrency and Scalability

4.1. Atomic operations

Targeting to an implementation with as few inter-thread locks as possible, all data structures accessed during the memory allocation procedure should optimally be accessed using atomic operations. An atomic operation performs an action without a need to fear for intervention by other threads, and without waiting. For good efficiency, direct support in the processor's instruction set is needed. Most modern processors implement a set of atomic instructions, most important of which are *compare-and-swap* and *load-linked/store-conditional. Compare-and-swap* is somewhat weaker, so it suffices to concentrate on that primitive – the *load-linked/store-conditional* can always be used to emulate the functionality of *compare-and-swap*. Figure 4 illustrates the operation of *compare-and-swap*. The whole procedure is executed atomically, with the system bus locked, so that other processors cannot break in.

```
COMPARE-AND-SWAP(location, comparand, value)
1   if VALUE-OF(location) = comparand
2         then VALUE-OF(location) ← value
3              return comparand
4         else return VALUE-OF (location)
```

Figure 4. The functionality of the *compare-and-swap* primitive.

The atomic primitive alone cannot do much. Appropriate data structures need to be built on top of it. As the previous chapter described, most of the data structures needed in the memory manager are linked lists. Implementing a general-use doubly linked list using only the *compare-and-swap* atomic primitive is not an easy task, although it can be accomplished [18]. Fortunately, in the design of the memory manager, this is not necessary – much simpler approaches suffice. They are outlined in the succeeding sections one by one as their use is discussed.

Hewlett-Packard has decided to offer an unpleasant surprise to developers trying to implement lock-free algorithms. In the HP PA-RISC architecture there is only one atomic synchronisation primitive: *load-and-clear*. This instruction atomically loads the contents of a memory location and zeroes it. The primitive is quite useless for anything more sophisticated than spin locks. Thus, critical data structure accesses have to be protected with spin locks, resulting in a negative effect on the performance. However, this thesis focuses on platforms with the *compare-and-swap* primitive available, leaving PA-RISC as material for possible future research.

4.2. Concurrent allocation

With multiple application threads allocating memory at their own pace, the first concurrency optimisation is targeted for simultaneous memory allocation. After the changes described in this section, garbage collection is still processed in a stop-the-world manner, but memory allocation does not require exclusion of other threads allocating memory.

As described in Section 3.4, the memory allocation procedure retrieves unused allocation units from the green object lists and, after some processing, inserts them to the black object list. These are the two data structures simultaneously used by all threads allocating memory. After a block has been removed from the green object list, there is no way another thread could gain access to it before it is again inserted in the black object list, so the operations performed on the allocation object itself need not be protected.

Green objects need only be removed from and inserted to the start of the green object list; thus, it can be changed from a doubly linked list into a singly linked stack with no trouble. Extracting the head of a singly linked list is easy to accomplish atomically. The algorithm is given in Figure 5.

```
SLL-EXTRACT-HEAD(list-head)
1   repeat
2       head ← list-head
3       if head = NIL
4           then ▷ The list is empty.
5               return NIL
6       next ← NEXT(head)
7   until COMPARE-AND-SWAP(list-head, head, next) = head
8   return head
```

Figure 5. The atomic singly linked list head extraction procedure using the *compare-and-swap* atomic primitive.

The algorithm relies on the fact that objects removed from the list are not reinserted into it during the execution of the procedure. While this condition holds, the execution is freed from a common problem of lock-free linked lists known as the *ABA problem* [18], resulting from a list element being removed and then added back to the list. However, this must be kept in mind when designing the sweep phase of the garbage collector; only there the objects are reinserted to the green object lists.

When the green object list is empty, a new macro block needs to be allocated. Fresh allocation units from the new macro block are inserted to the green object list. Atomic list insertion procedure is shown in Figure 6. The ABA problem discussed in the previous paragraph does not apply here, because the inserted items are fresh, never seen in the heap before. Should two threads running on two processors enter the new macro block allocation procedure simultaneously, two macro blocks are allocated, which is also not a problem – some extra space is just allocated for future needs, with the objects interleaved in the list.

The problem of parallel insertion of the allocation units into the black list is handled a bit differently. The black list must be doubly linked, because the collector needs to remove objects from random locations in the list. Instead of trying to implement a lock-free insertion procedure for the allocation units, each thread running in the system is given its own sandbox in the black object list. Basically, each thread's dedicated data structure contains the head of the thread's black object list, in which only that thread has access. Thus, need for a lock of any kind is removed.

```
SLL-INSERT-HEAD(list-head, item)
1   repeat
2         head ← list-head
3         NEXT(item) ← head
4   until COMPARE-AND-SWAP(list-head, head, item) = head
```

Figure 6. The atomic singly linked list head insertion procedure using the *compare-and-swap* atomic primitive.

The third shared object used by the memory allocation procedure is the heap size counter that reflects the need of a garbage collection. This counter is simply grown with the size of the new allocation unit. Implementing the addition with *compare-and-swap* is trivial.

Earlier, all memory management functions – both allocation and collection – were protected with a single mutual exclusion object. As several allocator threads now need to be allowed execution in parallel, this simple approach is not sufficient anymore. Instead, a semaphore is utilised. The semaphore's maximum count is set to the number of threads running in the system and increased and decreased in operating system originated callbacks as threads are born and die. Each allocation call starts with the acquisition of one semaphore resource and ends with its release. As such, all threads can run allocations in parallel. When garbage collection starts, the collection function reserves all the semaphore's resources, effectively blocking any threads from entering allocation routines. After this, the threads can be safely suspended – no thread except the collecting thread is in a memory management function, so the heap data structures are in a stable state.

4.3. Concurrent allocation of large objects

Large objects – larger than one macro block – need special attention, again. As described in Section 3.4.3, green large objects are held in a single singly linked list, sorted by their positions in the heap. In the concurrent allocator version the first scan for an exact match in size is removed: The allocation routine simply walks through the green object list, searching for the first green block of at least the required size.

The situation raises several problems. Even simply walking normally through the list is not safe: Thread B may remove the current block just before thread A accesses the block's *next*

pointer for advancing in the list. There is no way a simple singly linked list could be made safe in this respect.

Fortunately, the problem can be solved fairly easily by introducing the concept of *auxiliary nodes*. The basic idea can be used to implement fully lock-free doubly linked lists [18], but the structure used here is somewhat simpler. Auxiliary nodes are interleaved between the real nodes in the list so that there are never two adjacent real green objects in the list. Ensuring this is easy, since the only place where new objects are added to the list is during garbage collection. In the sweep phase, with other threads safely suspended, auxiliary objects can be created and placed in the list. Removal of objects from the list may cause two or more auxiliary nodes to become adjacent in the list, but this is not a problem.

Furthermore, all the real objects in the green object list are defined to have their *prev* pointers point to themselves, while the auxiliary nodes' pointers point to *nil*. Thus, it is easy to distinguish real nodes from auxiliary ones, and the distinction stays also after the real nodes are removed from the green object list: They are then inserted into a doubly linked, circular black object list, so the *prev* pointer cannot become *nil*.

When walking through the large object list, an iterator with two pointers is used. One of the pointers always points to an auxiliary node, while the other designates the real allocation node currently being examined. The auxiliary node is always the one immediately preceding the real node. With the list started by an auxiliary node, the first real node is found with the algorithm in Figure 7. The algorithm returns an iterator with both elements correctly set.

```
LG-NEXT-FROM-AUX(aux)
1   x ← aux
2   repeat
3       aux ← x
4       x ← NEXT(aux)
5       if x = NIL
6           then return { NIL, NIL }   ▷ List exhausted.
7   until PREV(x) ≠ NIL
8   return { aux, x }
```

Figure 7. Given an auxiliary node, this algorithm finds the next real node in the large green object list.

With an iterator pointing to a node in the list of green objects, the node can be inspected for suitability. If the memory block is sufficiently large, the node can be used to fulfil the current allocation request. First, the object is reserved by painting it black. The painting operation is done using the *compare-and-swap* operation so that no two threads can simultaneously reserve an object. The routine is given in Figure 8. Success of the reservation can be seen from its return value.

```
LG-PAINT(node)
1   if COMPARE-AND-SWAP(COLOUR(node), GREEN, BLACK ) = GREEN
2       then return TRUE
3       else return FALSE
```

Figure 8. An algorithm for safely painting a green object black.

If the reserved block is too large for the new allocation, it is split in two. The latter part is used for the new memory block and the original block, with a reduced size, is repainted green. However, in case the block is exactly the size of the new block, the original node is removed from the green object list and directly used for the allocation. The node has been reserved and nodes on both sides of it are auxiliary, so removing the node is easy: The *next* pointer of the iterator's auxiliary node is bent to the next auxiliary node.

Due to the imminent removal of nodes, walking the large green object list is not as trivial as following the *next* pointers. Instead, validity of all real nodes on the way needs to be ensured before moving on. The algorithm for doing this is shown in Figure 9. Since the *next* pointer of a node is only modified after the node has been removed from the list by twisting the preceding auxiliary node's *next* pointer to the next auxiliary node, the routine always calls *LG-Next-From-Aux* with the succeeding auxiliary node.

There is a downside to the green object list presented here. Between collections auxiliary nodes are not removed from the list, so they build up in groups as the real nodes dissolve, being used for allocations. As no new auxiliary nodes are added before the next collection cycle, the problem is not that bad. It is bad enough for a remedy to end up on the wish list for the next version, though.

```
LG-NEXT( { aux, node } )
1    x ← NEXT(node)
2    if NEXT(aux) = node
3         then return LG-NEXT-FROM-AUX(x)
4         else return LG-NEXT-FROM-AUX(aux)
```

Figure 9. The algorithm for finding the next valid iterator in the large green object list.

4.4. Concurrent collection

4.4.1. Concepts

The concurrent version of the garbage collector runs on its own dedicated thread parallel with the application threads. From now on, the other threads are called *mutator* threads: They access the heap objects and alter their contents, maliciously hindering the attempts of the collector. New memory allocations are also made in the mutator threads; allocation functions can be thought of as mutators, too. Thus, a heap object can simultaneously be accessed by several mutators, or by several mutators and the collector, but not by several collectors – there is only one.

All of the garbage collection cannot be performed in parallel with the mutators. More precisely, the very beginning and end of the collection cycle must be done in the stop-the-world mode where the collector knows that no mutators can mess with the heap. However, compared to the total duration of the garbage collection cycle, these stop times are brief. It is also possible to totally eliminate stopping mutators [11], but this enhancement is not in the scope of this thesis.

Parallel garbage collection brings in a new, big problem: After an object is coloured black, a mutator thread might write to the object and alter its contents. If this happens with a pointer being copied from a white object to a black one, and then deleted from the original position, it is possible that the collector never sees the pointer, as shown in Figure 10. Obviously, this must be prevented. Thus, writes to black objects must be trapped; moreover, writes to the grey object currently being scanned must be trapped as well. This must be achieved with the functionality offered by the underlying operating system and hardware platform.

On most modern systems, this boils down to write protection of whole memory pages, and handling write protection faults by marking a page *dirty* and unprotecting it, then resuming the execution at the point where the fault was signalled. Dirty pages need to be rescanned at a later stage.

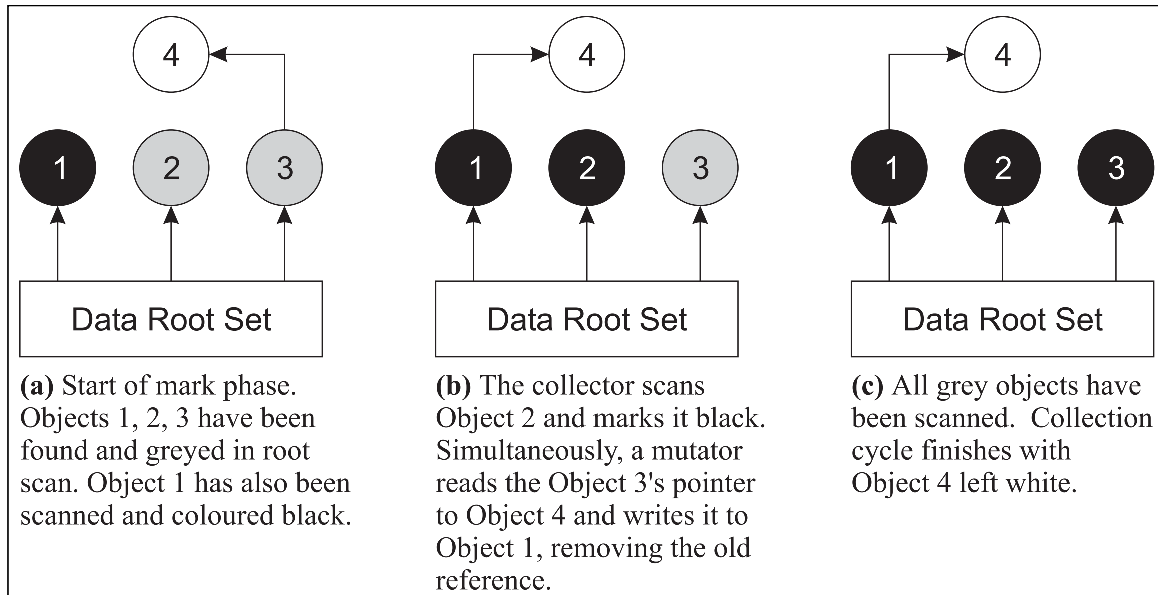


Figure 10. Illustration of a process resulting in a referenced object being declared garbage.

Some processors natively support dirty pages. That is, the processor automatically sets a bit in the page table as soon as a page is written to. Operating systems use this information for their own purposes, and usually do not expose the dirty bits to applications. However, some systems do: For example, Windows XP provides applications with functions for checking for pages' dirty bits as well as clearing them [13]. On systems with this support, the collector's performance is greatly enhanced, write protection of heap pages being unnecessary.

The per-page dirtying scheme has a downside: All allocation units residing on a page are simultaneously either dirty or clean. Thus, a page is marked clean when the first of its allocation units is scanned and made black. Actually, the memory manager handles macro blocks, possibly spanning several memory pages. Anyhow, if, after a macro block's first allocation object has been scanned, some other allocation object in the same macro block is written to, all black allocations in the macro block need to be rescanned.

Also, on Windows 2000, yet another problem arises when implementing page protection based guards: The exception handler installed to unprotect protected pages and mark them

dirty does not affect all system calls. Furthermore, some calls such as *ReadFile* check the destination buffer before filling it. As the destination buffer is write-protected, the function simply returns an error code, without reading anything. To prevent this from happening, the affected pages need to be locked up for the duration of call. It is not hard, but requires two new functions to the memory manager's interface. The first function grabs the semaphore, marks the macro block's pages clean and unprotects them. The system function can now be called. After it returns, the second function is called to mark the page dirty and to release the semaphore. In the future, a more elegant solution is desired, though.

In parallel collection, the colour of a newly allocated block also needs reconsideration. In the earlier stop-the-world collection model, new blocks were made black, to ensure there were only black and green objects in the heap at the point collection started. In case of concurrent collection and allocation, were new blocks painted black, two big problems would arise. First, it is quite natural that the code allocating memory immediately initialises the memory block after acquiring it. If the new block were black, this would immediately result in the page being marked dirty. Second, most objects are usually very short-lived [12]. If an object were painted black in an allocation made during a collection cycle, the said object would be guaranteed to live at least until the next collection cycle ends. If, instead, the object were painted white, and never seen again during the collection – as is the case with short-lived temporary objects that are only referred to through a thread's stack – the memory could be reclaimed immediately, at the end of the current collection cycle. As such, with concurrent collection, new allocations are made white.

4.4.2. Modifications to data structures

The heap structure must be altered before garbage collection can efficiently run in parallel with the mutators. First of all, the allocation unit headers must be decoupled from the allocation units they represent. This decoupling allows modifications to be made to the headers even when the actual allocation page is marked read-only, for trapping mutators' write attempts to blocks that have already been scanned. The headers of the allocation units are allocated in arrays on separate macro blocks in the heap and the pointer to the header array is included in the macro block header structure. The algorithm of mapping a pointer candidate to the corresponding unit header gets a new step where the header array is indexed with the allocation unit's order number. The lines modified from the original algorithm of Figure 2 on page 19 are listed in Figure 11.

```
MAP-POINTER(candidate)
. . .
12  index ← ⌊ (diff − MACRO-BLOCK-HEADER-SIZE) / ALLOCATION-SIZE(mbheader) ⌋
13  if index ≥ ALLOCATION-COUNT(mbheader)
14        then ▷ "Leftover" area
15              return NIL
16  header-array ← HEADER-ARRAY(mbheader)
17  alloc ← header-array[index]
18  if COLOUR(alloc) = GREEN
19        then return NIL
20  return alloc
```

Figure 11. The second vesion of the function for mapping a pointer into the corresponding allocation block, modified for concurrent collection.

The allocation unit header arrays naturally need to be dynamically allocated. They are allocated from the same heap with the actual allocations. To make thread safety issues easy, each mutator thread has its own set of header array macro blocks. New header arrays are allocated from these macro blocks. As the heap never shrinks, header arrays are never freed. Thus, allocation is trivial, and after the allocation all threads are free to access the contents of the header array.

With the colour of newly allocated blocks changed to white, the block lists must also be adjusted a bit. In the new version there is only one black object list, but separate white object lists for each thread. Each thread adds new blocks to a white object list of its own, and the collector thread is fully in control of the black object list.

Additionally, two bit arrays with a bit for every macro block are introduced for handling the write protection of scanned pages. One of the arrays stores the dirty bits for the macro blocks. They are set when a write protected macro block is written to. The other simply speeds up protection, tracking pages that are currently protected: By checking this table before calling the operating system's page protection function unnecessary system calls can be avoided.

4.4.3. Garbage collection cycle

Like the original, non-concurrent collection, the collection starts when the heap size exceeds the previous collection's result size by a constant factor. A dedicated garbage collection thread kicks in and first stops all other threads in the process.

Swapping the black and white colours is moved from the start of collection cycle to the end of it. As such, between collections, there are now only green and white objects. At the beginning of a collection cycle, the collector now simply searches for the data roots and scans them. The roots could as well be just stored and scanned later while the mutators are running, but that approach would need additional buffering for the data roots' areas. As the roots only form a small portion of the process's data, scanning them does not take long and the extra buffering can be forgotten, making the implementation a bit more straightforward.

After the data roots have been scanned, the mutator threads are resumed from their suspended state. Also, the memory allocation semaphore's resources are relinquished, so that memory can again be allocated.

During the scanning of the data roots some of the heap's allocation units usually get marked grey. In fact, it is possible that none is – in this case, none of the heap's objects is alive and the collection ends immediately. Usually, though, some objects are accessible from the data roots and are greyed. These grey objects are scanned next.

When a reference to a white object is encountered, the object must be removed from its list, coloured grey and placed in the collector's grey object list. As described earlier, each thread has its own white object list, into which newly allocated objects are inserted. Now, the garbage collector thread must also remove objects from these lists; furthermore, it will access the lists in a random order – in the order references to white objects are found. For this reason, the list must be doubly linked.

It would be possible – yet laborious – to build the doubly linked white object lists with auxiliary nodes [18]. Luckily, the fact that mutator threads only need to insert objects at the head of the list simplifies the case enough to warrant another approach. With carefully planned functions for attaching and detaching objects, both allocators and the collector can safely use the list.

The routine for attaching an allocation object at the head of a white object list is shown in Figure 12 and the corresponding routine for detaching an arbitrary white list object can

be seen in Figure 13. The algorithms effectively use the *prev* pointer of the list's first item as a spin lock. Usually, spin locks are not good for performance, but this lock is actually needed very rarely: The Compare-And-Swap operations in the algorithms only fail when the collector tries to remove the very first object from a white object list with the corresponding mutator thread simultaneously inserting an object to the list.

```
WL-ATTACH(head, item)
1   PREV(item) ← NIL
2   repeat
3       x ← NEXT(head)
4   until COMPARE-AND-SWAP(PREV(x), head, NIL ) = head
5   NEXT(head) ← item
6   PREV(x) ← item
7   NEXT(item) ← x
8   PREV(item) ← head
```

Figure 12. An algorithm for attaching an allocation object to the head of a white object list.

```
WL-DETACH(item)
1   repeat
2       prev ← PREV(item)
3   until prev ≠ NIL ∧ COMPARE-AND-SWAP(PREV(item), prev, NIL ) = prev
4   PREV(NEXT(item)) ← prev
5   NEXT(prev) ← NEXT(item)
6   return item
```

Figure 13. An algorithm for detaching an allocation object from a white object list.

When a grey object becomes black, the memory belonging to the object must be protected against mutators' writes. The protection must be initiated before starting to scan through the grey object – otherwise, the block might change just as it is scanned and the change go unnoticed. Still, it is possible that the block changes just as the collector is processing it, but that does no harm – the change is noted and the block scheduled for later rescan.

The processing of the grey object list proceeds in parallel with mutator threads. When the list becomes empty, the dirtied pages are rescanned as described earlier. This, too, can be done in parallel with mutators. Scanning the dirty pages possibly results in new grey objects whose references were found, and so the grey object list is processed again. This loop continues until there are no new grey objects and no new dirty pages, or the collector determines that mutators are changing certain pages' contents all the time as the collector tries to keep up to date with them. Generally, if the number of dirty pages does not significantly decrease during a loop pass, it is time to end the collection.

To optimise the scanning process, grey objects on macro blocks already marked dirty need not be scanned. There is no sense in scanning them, because they would be rescanned anyway, along with other black objects on the macro block. Instead, they can be directly painted black.

After both the grey object list and the dirty macro block list are exhausted, the garbage collection cycle is ready to complete. The world is again stopped, as at the beginning. As new references to objects may have appeared in the data roots since the start of the collection – the roots were not write-protected – the roots need to be rescanned. This operation should not produce too many references to white objects, so it makes sense to also scan the newly found objects with the mutator threads suspended. Finally, the remaining white objects are declared garbage and swept to their respective green object lists, write protected pages are unprotected, the white and black colours are swapped, mutator threads resumed and the collection cycle ends.

The final touch of concurrency can be added by eliminating the initial stopping of the world. This is possible because at the beginning the roots need not be stable – they simply give a starting point for the collection. Thus, the only remaining stop time occurs at the end of collection, where a static image of the heap is required to ensure that all heap references are correctly traced.

4.5. Basic performance optimisations

The most straightforward way of starting garbage collection is to simply use a flag. The flag is set in the memory allocation function when the heap has reached a predetermined threshold value. When the flag is raised, the dedicated garbage collection thread activates and starts a collection cycle. After collection ends, the flag is reset and the threshold value is recalculated for the next collection cycle.

With this kind of an arrangement, an application that allocates plenty of memory may cause problems: Memory allocation is always much faster an operation than the corresponding collection work, so for an application simply sitting in a loop allocating memory it is easy to outperform the collection. The collector never has the time to process all allocated memory before more is allocated, so collection cycle never ends and the application finally runs out of memory.

The easiest solution for this problem is to define another threshold value. After the heap's size reaches this second threshold, new memory allocation is not allowed until the currently running collection cycle completes. Another flag is used here: The allocating thread waits until the flag is set, and the collector thread sets the flag when completing collection.

However, more sophisticated approaches are also available. On multithreaded operating systems, each thread has a priority value. Threads with higher priority get executed more often and for longer times than those with low priority. As such, the priority of the garbage collection thread could be adjusted during the collection at several threshold values. For example, the collector could start running at the lowest priority available, thus only getting processor time when other threads are idle. If there is not enough idle time, and the heap keeps growing, the collector thread's priority would be raised gradually until it gets all the CPU time it needs.

Even this kind of priority control may not be sufficient. On computers with several processors, one or more allocator threads will get run full-time on other processors as the collector only utilises one thread and one processor. The only way to help this situation is to devise several collection threads – for example, one per processor. This kind of a system would be much more complicated, and it is out of the scope of this thesis.

Another clear point of optimisation, at least on Windows 2000 systems, is the use of the operating system's semaphore objects. They seem to be terribly inefficient: In a quick performance test, the wait and post operations on the semaphore when allocating memory

took, on average, as much CPU time as the rest of the allocation function. As this is quite unacceptable, a better exclusion scheme must be devised.

Actually, the semaphore object is already used in a rather non-semaphore-like fashion. Each thread has its own access ticket in the semaphore, and they only need to wait on the semaphore when the collector thread is stopping them. The semaphore is now discarded and a state variable is allocated instead for each thread. Each thread can be in one of the five states: *Running, Allocating, About To Stop, Waiting for Allocation* or *Waiting for Collection.* The state graph is presented in Figure 14.

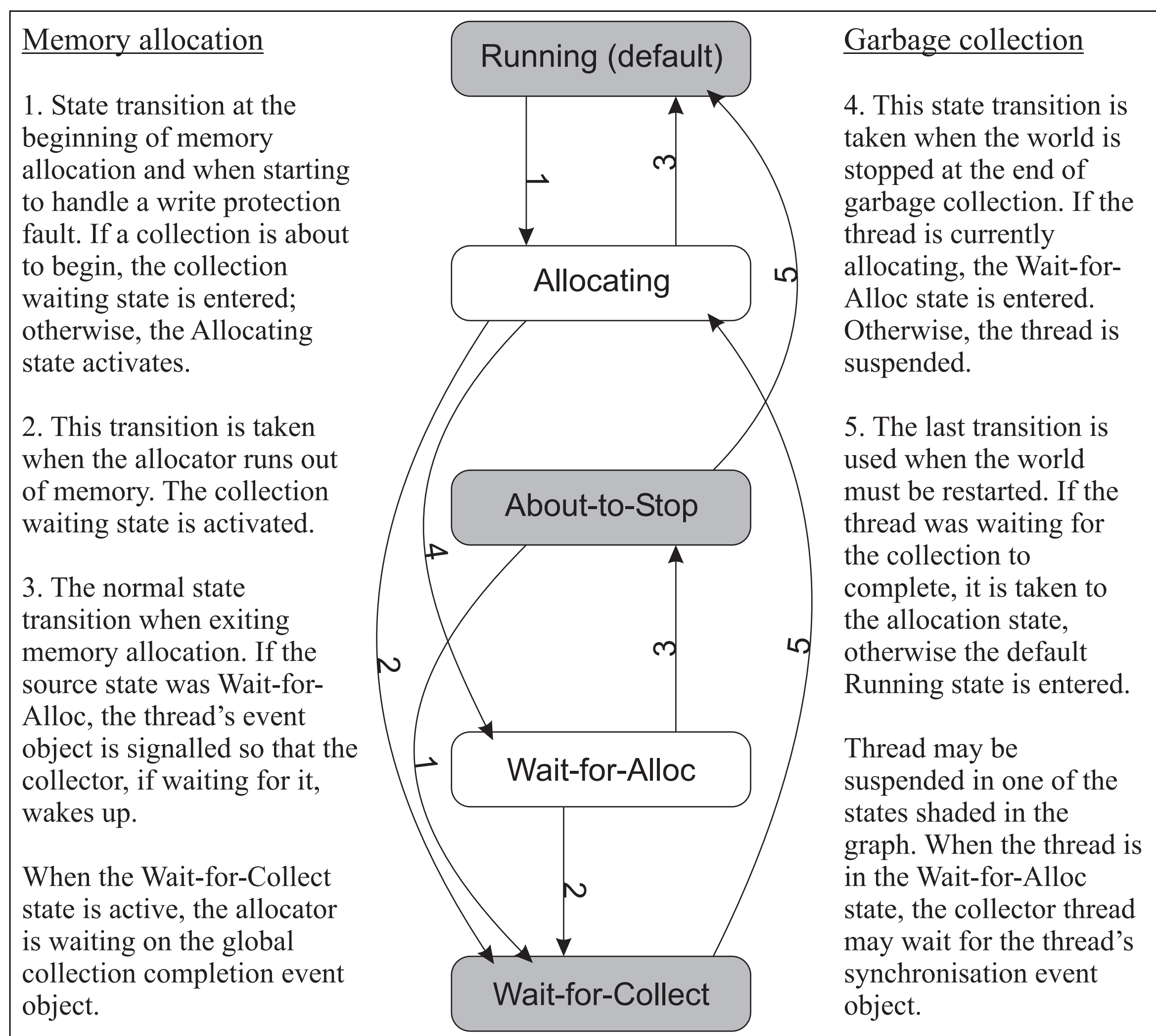


Figure 14. State graph for thread state variables.

When a new thread starts up, it begins in the Running state. This state is is active whenever the thread has nothing to do with memory management. The Allocating state is active when

the thread is executing the memory allocation functions. If the thread runs out of memory, it enters the Waiting for Collection state, where it waits until the next collection ends and then resumes the allocation attempt. If the collector finds a thread in the Allocating state when it should be stopped, it switches the thread to the Waiting for Allocation state. When exiting the allocation in this state, the allocator routine signals the thread's completion flag and enters the About to Stop state. If another allocation is attempted in the About to Stop state, the thread is immediately suspended.

The state variable can be controlled fully with the atomic compare-and-swap operation. Thus, operating system-defined functions need only be used when either the allocator really needs to wait for collection to complete or the collector needs to wait for an allocator to finish. Each thread has an event object on which the collector can wait when the thread is allocating memory; the collector has a single global event object that signals end of collection. All the allocating threads can wait on this object.

5. EVALUATION

5.1. Concepts

In this chapter, the first implementation of the memory manager described in this thesis is called *IonMemory* – pronounced *eye-on-memory*.

IonMemory is compared to two other memory management implementations. The standard memory manager provided with Microsoft Visual C/C++ 6.0 runtime is a standard C++ memory manager, without garbage collection functionality. It is a very streamlined implementation with good multiprocessor behaviour. The Boehm Garbage Collector library [9] version 6.1 is used as the other comparison point. The library, being the industry standard in C/C++ conservative garbage collectors, has been heavily tuned to work as swiftly as possible. It includes incremental collection support, which is benchmarked to also get a reference point in the concurrent garbage collection category, bringing the number of different configurations up to four.

Several variables were measured with each of the managers, on each test case. Most important of them is, naturally, the application's execution time. As all tests are run on a two-processor computer, another interesting number is the relative use of CPU time: This gives some idea of both the application's and memory manager's scalability to multi-processor systems. The third important number is the application's maximum consumption of memory.

All tests were run on an Intel Pentium II computer with two 550 MHz processors. The computer had 512 megabytes of RAM, out of which at least 256 megabytes was available to the running test application. Windows 2000 was used as the operating system, and performance measurements were taken with the operating system's own functions.

The goal of the evaluation was not to authoritatively classify the compared memory managers. Merely, the tests were used to ensure the overall feasibility of the design choices made in the development of IonMemory and to prove the concept of a parallel, concurrent, scalable, conservative garbage collecting memory manager viable. Also, the results are meant to give directions about where to head for further development and optimisation of IonMemory.

5.2. Case study 1: Multithreaded memory allocator

A simple application that allocates chunks of memory into a linked list was used as the first test case. The application stores a list of pointers to variably sized memory blocks, fills the blocks with sequential numbers and before letting them go checks that the blocks are unchanged. Actually, the same application was used as test software when implementing IonMemory.

The application can be easily made multithreaded; each thread keeps its own list of allocations. Test runs were performed with one to eight threads. This gives a good estimate of the scalability of the memory managers. However, the allocated blocks are never seen or handled by threads other than the one that allocated them, which makes the threading situation a bit less realistic.

During the run of the application, each thread performed about 1.2 million allocations averaging 667 bytes in size. The total amount of allocated memory was 800 megabytes per thread. Only up to two hundred allocations per thread were accessible at any time, though: Thus, the effective working set averaged only 130 kilobytes per thread. The theoretical maximum of a thread's effective heap space consumption was about 782 kilobytes.

The test application was run with one to eight threads. Each run was repeated five times and the results averaged. The results are shown on Table 3. Not surprisingly, the manual memory management of Visual C gives the best numbers. It uses least time, both CPU and real time, and exhibits excellent scalability: With two threads, the application uses 97% of the two processors' time. The Boehm collector does not scale that well – aside of the two performance peaks at two and six threads – and is also considerably slower than Visual C, but its memory performance is quite good. The maximum memory use is only double of Visual C's. The incremental version scales a bit better and more consistently, but it is also much slower.

Table 3. Multithreaded memory allocation test program results.

CPU time (seconds)

Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Visual C	7.4	21.3	33.1	45.2	69.0	119.8	144.0	165.1
Boehm/Stop	29.0	65.2	107.5	173.1	249.2	290.0	344.1	390.7
Boehm/Incr.	46.5	101.9	183.4	248.8	314.4	366.5	437.6	494.1
IonMemory	27.0	53.4	86.9	115.8	145.5	173.0	200.9	228.4

Real time (seconds)

Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Visual C	7.4	10.9	16.9	23.0	38.2	72.7	87.7	100.2
Boehm/Stop	29.0	51.6	99.9	162.5	233.7	223.3	319.8	360.8
Boehm/Incr.	46.4	84.7	163.9	226.5	281.2	327.1	387.7	436.0
IonMemory	22.8	34.5	54.5	73.8	89.9	108.0	123.2	142.5

CPU use (per cent on a double-CPU machine)

Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Visual C	49.8	97.4	98.3	98.3	90.3	82.4	82.1	82.4
Boehm/Stop	50.0	63.1	53.8	53.3	53.3	65.0	53.8	54.1
Boehm/Incr.	50.0	60.1	55.9	54.9	55.9	56.0	56.4	56.7
IonMemory	59.0	77.4	79.7	78.5	80.9	80.1	81.5	80.1

Maximum memory use (megabytes)

Threads	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Visual C	1.5	2.1	2.7	3.3	3.8	4.6	5.1	5.8
Boehm/Stop	2.7	3.6	5.3	6.6	6.6	8.4	9.3	10.7
Boehm/Incr.	3.0	4.2	5.3	6.4	7.5	8.9	9.7	11.1
IonMemory	6.6	11.2	15.6	18.8	24.4	26.2	29.3	33.5

IonMemory does pretty well in the first test case. This is perhaps not a big surprise, though, the test case being the software's own test application. It beats the Boehm collector clearly in the timings and scales well, consistently using about 80% of CPU time. On eight threads, the Visual C implementation uses 13.6 times the real time of a single thread, while on IonMemory the ratio is 6.2.

The memory behaviour of IonMemory is not as good as its time performance, though. It devours about three times the memory of the Boehm collector and five times the memory of Visual C. It is not clear why this happens. However, the use of memory is relative to the number of running threads and thus the size of the actual live heap, *not* to running time. This means that the garbage collector is working correctly. Also, changing the garbage collection into a stopping collection did not affect the numbers considerably: As such, the space is probably due to inefficiencies in the memory manager's memory structures.

5.3. Case study 2: GNU AWK

In an earlier study of conservative garbage collection performance, Benjamin Zorn used six applications to benchmark memory managers [23]. Of these, GNU AWK (gawk) was easiest to come by – and compile on Windows 2000 – so it was selected as the second application for the performance evaluation.

AWK is a simple programming language with powerful regular expression based text processing capabilities. A test run thus requires both an AWK source file and an input file for processing. The simple *adj* script that indents and aligns text files was selected as the source file. The script is presented in the book "sed & awk" [16, Section 13.5]. The script was instructed to fill 72-character justified lines with a four-space indent at the left edge. The input file was an English language text file of about 40 000 lines and 1.7 megabytes.

Each of the four configurations ran the script through five times. The average values from these runs are shown in Table 4.

Table 4. GNU AWK test run results.

Manager	CPU time (s)	Real time (s)	CPU use (%)	Max. mem. (MB)
Visual C	50.5	50.6	49.9	0.9
Boehm/Stop	61.4	61.6	49.8	2.2
Boehm/Incr.	109.0	109.4	49.8	2.1
IonMemory	97.7	85.3	57.3	2.8

The results show that Visual C is again the fastest and also most memory efficient of the implementations. The stopping version of the Boehm collector is second fastest, the concurrent collectors coming last. IonMemory wins the incremental Boehm collector by 24 seconds in real time, even though in CPU time the difference is only 11 seconds. This results

from the fact that garbage collection can be performed in parallel with the actual work on the spare processor.

IonMemory uses a bit more memory than the Boehm manager; both of them lose significantly to Visual C. The working set of the application is very small, and thus the inefficiencies in IonMemory's use of memory do not show too badly in these results.

5.4. Case study 3: Boolean Expression Reduction

The third test case is an application that reduces Boolean expressions into simpler ones. The application was created specifically for the test case. The application takes in an arbitrary Boolean expression containing *and, or* and *not* operators and a number of variables. The expression is then expanded into disjunctive normal form that is simply a disjunction consisting of conjunctions that in turn only contain variables and negated variables. This format is then searched for tautologies and contradictions, after which the – hopefully – simplified formula is restructured into a tree, using operator distributivity and De Morgan's laws. The process is shortly illustrated in Figure 15.

$A \wedge \neg((\neg B \vee (\neg A \wedge B)) \wedge C)$	(Original expression)
$\equiv (A \wedge A \wedge B) \vee (A \wedge \neg C) \vee (A \wedge B \wedge \neg B)$	(Disjunctive normal form)
$\equiv (A \wedge B) \vee (A \wedge \neg C)$	(Optimised version)
$\equiv A \wedge (B \vee \neg C)$	(Restructured tree)

Figure 15. Illustration of operation of the Boolean reductor.

Expansion of an arbitrary Boolean expression may be costly: The size of the expression may grow exponentially. This naïve implementation does not do any sanity checks and can thus consume very much memory. However, with conservatively enough chosen parameters for generating the input data, the application served well as another more or less real-world application with which to benchmark the memory managers.

In the tests, the application processed ten complex Boolean expressions. The expressions were identical for all runs and for all memory managers and they had a total of a million operators and variables. During each run a total number of 47 million allocations were made, with a total allocation load of 1.2 gigabytes. The average allocation size is very small, only 25 bytes. Run statistics for the memory managers are shown in Table 5. The test was run with one and two threads.

Table 5. Boolean Reduction test results.

One thread

Manager	**CPU time (s)**	**Real time (s)**	**CPU use (%)**	**Max. mem. (MB)**
Visual C	205.9	206.8	49.8	64.6
Boehm/Stop	85.7	86.1	49.8	17.1
Boehm/Incr.	108.3	108.9	49.7	19.0
IonMemory	158.1	126.1	62.7	67.3

Two threads

Manager	**CPU time (s)**	**Real time (s)**	**CPU use (%)**	**Max. mem. (MB)**
Visual C	193.6	123.8	64.6	78.2
Boehm/Stop	200.4	155.7	64.4	17.1
Boehm/Incr.	250.3	196.5	63.7	20.5
IonMemory	372.7	217.0	85.9	74.5

The results differ much from those of the previous two applications. With one application thread, Visual C is now, surprisingly, the slowest one. This is probably due to the fact that the application was designed for a garbage collecting memory manager and does not care too much about memory leaks – as such, Visual C's heap grows much larger than the actual working set, which makes the manager ineffective. The Boehm collector does very well on one thread, and IonMemory uses a lot of memory, as before.

On two threads, the situation changes. Visual C speeds up considerably, while the garbage collecting managers slow down. The CPU time use of all three managers grows by about 130%, but better parallel processing reduced the effect somewhat. Real time use of the Boehm collector grows by 80% and IonMemory's by 72%.

Some research was done to find the reason for this slow-down of automatic memory managers. The research was done with IonMemory, and the reason was easily located. With one thread, there were 550 write protection faults from an application thread writing on a page protected by the memory manager; with two threads, the number of write protection faults was about 250 000. For each fault, the corresponding memory page was unprotected, and then later protected again by the memory manager. This consumes very much time, and results probably from relatively longer garbage collection times when two mutator threads are running simultaneously. Also, as the allocations are very small, many of them fit in a single

macro block and a write to one of them is enough to result in rescanning the whole macro block. The same reason probably applies also at least to the incremental Boehm collector that uses a similar page protection scheme.

5.5. Summary

Test results clearly state the problematic areas in IonMemory. It is easy to focus further development and optimisation work to them. More precisely, heap structure should be closely analysed and the core reason for the IonMemory's huge use of memory located. Another important focus point is the inefficiency of the dirty-page marking scheme, as discovered in the third test case. These improvements are, however, way out of the scope of this thesis.

On the other hand, the results also show the competitiveness of the designed memory manager, when compared to two state-of-the-art memory managers that have undergone much optimisation work. Also, the selected data structures are seen to work as planned, because even in a single threaded application IonMemory is able to utilise both processors for memory management. This is a feature not found in the other two memory managers and is rare also elsewhere in the memory management world.

6. Related Work

6.1. Garbage collection and C++

C++ is not one of the easiest languages for which to design a garbage collecting memory manager. Data in a C++ program is very weakly typed: Same memory location can be interpreted in any way a programmer wishes by simply pointing to it through differently typed pointers. Also pointer arithmetic is supported and encouraged, which makes it even harder to sieve the correct pointers out of the huge mass of candidates. However, as stated in Chapter 1, there are two good solutions to the problem that are available to the public.

John F. Bartlett's collector [4, 5] is limited. The collector assumes cooperation from the application: The collector must be explicitly told about collectable objects and their use. On the other hand, the collector is quite powerful, because it knows of the objects and can easily track them. It is thus well suited for some uses, such as implementation of an interpreter of a garbage-collected language.

More related to this thesis, Hans J. Boehm has implemented a general-use garbage collector for C and C++ [6, 9]. This implementation can be used in virtually every C or C++ program simply by adding the collector's include files to the project's files and recompiling the application. As with the work described in this thesis, all calls to memory allocation functions are replaced by calls to the garbage collector's functions and freeing functions are replaced with null functions.

It is important to note that conservative collectors, like the Boehm collector can also be used with languages that do support run-time type information and garbage collection. For example, the popular PLT Scheme environment uses the Boehm collector by default [14]. Using a conservative collector enables the language to define a foreign function interface

easily, without needing to worry about memory management of objects exposed to foreign language functions.

The Boehm collector also acted as a motivator for this work. Many solutions are similar in these two collectors, but parallelism is one of the features the Boehm collector does not support well.

For conservative collectors, the application programmer should also select the data structures used in the program carefully. As data roots are ambiguous, obsolete pointers may be left to point to data structures that are actually garbage. By selecting data structures properly, the negative effects of this functionality can be significantly reduced. Boehm has published an excellent article on the subject. [8]

6.2. Parallel collection

There is a parallel garbage collection extension available to the Boehm's collector [20]. The approach taken in the extension is to simply make the collection work parallel. There are several dedicated garbage collection threads – one per processor – that perform the collection in one piece while application threads are stopped. This is efficient, but also means longer stop times than collection working concurrently with allocation.

Boehm himself, together with Alan J. Demers and Scott Shenker, has also devised a parallel, concurrent collection algorithm for the collector [7]. That algorithm, however, is somewhat specialised on a certain system and performs collection in two stages, only one of which is concurrent with allocation. Only a part of allocated objects is collected concurrently; a full stop-the-world collection is run occasionally to completely collect unreferenced objects.

For copying collectors, there are a variety of attempts on parallel garbage collection. During the last few years, apparently the appearance of Java has directed much attention to tuning garbage collection, especially on hardware used when running web-based software. One of the most interesting ones is Sapphire, designed by Richard L. Hudson and J. Eliot B. Moss, that claims to perform garbage collection totally in parallel, without the need to stop the world – that is, mutator threads – at all [11]. However, Sapphire, like other copying garbage collectors, relies heavily on the support provided by the language, and is thus less related to the work described in this paper.

7. Conclusions

In this thesis, a design was outlined with which a conservative, parallel memory manager can be constructed for C++. The manager is basically plug-and-play, requiring the programmer no changes in the program code. First, a basic implementation without the parallelism features was presented. The basic design was then extended into a fully functional parallel memory manager. Lock-free algorithms for handling the data structures necessary in memory management were devised and presented.

At the same time, an example implementation was built on top of the described design. The implementation, *IonMemory*, was benchmarked against two memory managers and the results presented and analysed. The implementation was found to justify its existence quite well, although it could not match the performance of the other two older and more fine-tuned managers. Its memory footprint was significantly larger than the others' and it was also somewhat slower, especially in more realistic application environments. However, the results indicated that memory management work was indeed successfully parallelised.

Currently, the prospects of automatic memory management seem good. Modern languages and systems include support and functionality for garbage collection, and even older languages are given their chances. Microsoft Visual Studio .NET defines an extension to the C++ language with which data can be tagged for automatic memory management. The Boehm garbage collector library comes bundled in the GNU C 3.2.2 installation package. Even Bjarne Stroustrup, the designer and original implementer of C++, has indicated his wish of automatic memory management to be acknowledged as a valid implementation technique for C++ memory management [17].

Naturally, languages with built-in garbage collection support do not need the conservatism features that greatly restrict the design of C++ garbage collectors, such as the one described

in this thesis. However, C++ still seems to be going strong, so most likely there is still going to be demand for C++ garbage collector libraries. After C++ programmers get a taste of automated memory management when trying Java or other garbage collected language they will be looking for similar memory management solutions for C++, too.

As for the future work concerning the parallel collector design presented in this thesis, there is a lot. Even though the first implementation works and exhibits reasonable performance figures, it is not yet really competitive. The idea of collection in parallel with the allocating and mutating threads seems to work, though. As such, development of the memory manager will continue.

The memory manager was also designed to be reasonably easily portable to other operating systems. This far, only Windows 2000 has been used as a development platform. At least Linux, and presumably some Unix systems such as HP-UX and Solaris, should be supported in a close future. Also, the manager should be able to take advantage of the advanced features of Windows XP and newer Microsoft operating systems. At least the control of hardware dirty pages given to the memory manager is likely to speed up parallel garbage collection considerably.

The memory manager is to be released with full source code free of charge for the use of anyone. The current version is not stable and robust enough to warrant this, but later the feedback from potential users of the manager is going to give indication as for where to head with the memory manager's development. Also, for a commercial product, it would be hard to compete against the existing free implementations.

References

[1] Baker, Henry G. List Processing in Real-Time on a Serial Computer. *Communications of the ACM* **21(4)**, 280–294 (April 1978).

[2] Baker, Henry G. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices* **27(3)**, 66–70 (March 1992).

[3] Baker, Henry G. Infant mortality and generational garbage collection. *ACM SIGPLAN Notices* **28(4)**, 55–57 (April 1993).

[4] Bartlett, Joel F. Compacting Garbage Collection with Ambiguous Roots. Tech. Rep. 88/2 (February 1988), DEC Western Research Laboratory, Palo Alto.

[5] Bartlett, Joel F. A generational, compacting collector for C++. *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, Ottawa* (October 1990).

[6] Boehm, Hans-J. and Weiser, M. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience* **18(9)**, 807–820 (September 1988).

[7] Boehm, Hans-J., Demers, Alan J. and Shenker, Scott. Mostly Parallel Garbage Collection. *SIGPLAN Notices* **26(6)**, 157–164 (1991).

[8] Boehm, Hans-J.. Space Efficient Conservative Garbage Collection. *ACM SIGPLAN Notices* **28(6)**, 197–206 (June 1993).

[9] Boehm, Hans J. A garbage collector for C and C++. URL *http://www.hpl.hp.com/personal/Hans_Boehm/gc/*. URL verified in May 2003.

[10] Dijkstra, Edsger W.; Lamport, Leslie; Martin, A.J.; Scholten, C.S.; Steffens, E.F.M. On-the-fly garbage collection: an exercise in multiprocessing. *Communications of the ACM* **21(11)** (EWD496), 966–975 (November 1978).

[11] Hudson, Richard L. and Moss, J. Eliot B. Sapphire: Copying GC Without Stopping The World. *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande (ACM Press)*, 48–57 (2001).

[12] Lieberman, Henry and Hewitt, Carl E. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM* **26(8)**, 419–429 (June 1983).

[13] Microsoft Platform Sofware Development Kit, April 2003.

[14] The PLT People. PLT Scheme. URL *http://www.plt-scheme.org/*. URL verified in May 2003.

[15] Rovner, Paul. On adding garbage collection and runtime types to a strongly-typed, statically-checked, concurrent language. Tech. Rep. CSL–84–7 (July 1985), Xerox Palo Alto Research Center.

[16] Dougherty, Dale and Robbins, Arnold. *sed & awk*. O'Reilly. 2nd Edition, March 1997. 429 pages.

[17] Tran, Pierre (interviewer). Interview of Bjarne Stroustrup by "Developpeur Reference". URL *http://www.research.att.com/~bs/nantes-interview-english.html*. URL verified in May 2003.

[18] Valois, John D. Lock-Free Linked Lists Using Compare-and-Swap. *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (ACM Press)*, 214–222 (1995).

[19] Wilson, Paul. Uniprocessor Garbage Collection Techniques. *Proc. Int. Workshop on Memory Management (Springer-Verlaag)* **637**, 1–42 (September 1992).

[20] Yonezawa Laboratory. Parallel and Distributed Extension to Boehm's Conservative Garbage Collector for C/C++. URL *http://www.yl.is.s.u-tokyo.ac.jp/gc/*. URL verified in May 2003.

[21] Zorn, Benjamin. Comparing Mark-and-sweep and Stop-and-copy Garbage Collection. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming (ACM Press)*, 87–98 (June 1990).

[22] Zorn, Benjamin. Barrier Methods for Garbage Collection. Tech. Rep. CU-CS-494-90 (November 1990), University of Colorado, Boulder.

[23] Zorn, Benjamin. The Measured Cost of Conservative Garbage Collection. *Software — Practice and Experience* **23(7)**, 733–756 (1993).

Appendix A. Interface header

```
/**
 * @file           ionmemory.h
 * @guid           f3faaf9f7b9c4e089d9e5b15c75358e7
 * @copyright      2002-2003 Ionific Ltd.
 * @author         Jussi Lahdenniemi
 * @project        ionmemory/include
 * @description    IonMemory library main header
 */

#ifndef Hf3faaf9f7b9c4e089d9e5b15c75358e7
#define Hf3faaf9f7b9c4e089d9e5b15c75358e7

#include <new>
#include <memory>

#if defined( _MSC_VER ) && _MSC_VER <= 1200
#   define longlong __int64
#else
#   define longlong long long
#endif

#ifdef _MSC_VER
#   define MEMEXPORT __declspec( dllexport )
#else
#   define MEMEXPORT
#endif

namespace IonMemory
{
   enum BlockType
   {
      Normal,
      NoPointer,

      BlockTypeMax
   };

   MEMEXPORT void* alloc( size_t, BlockType = Normal );
   MEMEXPORT bool lock( void const* );
   MEMEXPORT bool unlock( void const* );
   MEMEXPORT size_t size( void const* );

   MEMEXPORT std::bad_alloc& getBadAlloc();
```

```
template<class T>
class NoPtrAllocator
{
public:
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef T const* const_pointer;
    typedef T& reference;
    typedef T const& const_reference;
    typedef T value_type;

    pointer address( reference x )
    {
        return &x;
    }

    const_pointer address( const_reference x ) const
    {
        return &x;
    }

    void deallocate( void* ptr, size_type )
    {}

    size_type max_size() const
    {
        return ((size_type)–1) / sizeof( T );
    }

    pointer allocate( size_type n, void const* )
    {
        return new( IonMemory::NoPointer )T[n];
    }

    char* _Charalloc( size_type n )
    {
        return new char[n];
    }

    void construct( pointer p, T const& val )
    {
        new( (void*)p )T( val );
    }

    void destroy( pointer p )
    {}
};

} // namespace IonMemory

__inline void* operator new( size_t size )
    throw( std::bad_alloc )
{
    void* ptr = IonMemory::alloc( size );
    if( ptr == 0 && size ≠ 0 )
    {
        throw IonMemory::getBadAlloc();
    }
    else
    {
        return ptr;
    }
```

```
}

__inline void* operator new( size_t size, IonMemory::BlockType type )
   throw( std::bad_alloc )
{
   void* ptr = IonMemory::alloc( size, type );
   if( ptr == 0 && size ≠ 0 )
   {
      throw IonMemory::getBadAlloc();
   }
   else
   {
      return ptr;
   }
}

__inline void* operator new( size_t size, std::nothrow_t const& )
{
   return IonMemory::alloc( size );
}

__inline void* operator new( size_t size, std::nothrow_t const &,
                             IonMemory::BlockType type )
{
   return IonMemory::alloc( size, type );
}

__inline void* operator new[]( size_t size )
   throw( std::bad_alloc )
{
   void* ptr = IonMemory::alloc( size );
   if( ptr == 0 && size ≠ 0 )
   {
      throw IonMemory::getBadAlloc();
   }
   else
   {
      return ptr;
   }
}

__inline void* operator new[]( size_t size, IonMemory::BlockType type )
   throw( std::bad_alloc )
{
   void* ptr = IonMemory::alloc( size, type );
   if( ptr == 0 && size ≠ 0 )
   {
      throw IonMemory::getBadAlloc();
   }
   else
   {
      return ptr;
   }
}

__inline void* operator new[]( size_t size, std::nothrow_t const& )
{
   return IonMemory::alloc( size );
}

__inline void* operator new[]( size_t size, std::nothrow_t const &,
                               IonMemory::BlockType type )
{
```

```
    return IonMemory::alloc( size, type );
}

__inline void operator delete( void* )
{}

__inline void operator delete( void*, IonMemory::BlockType )
{}

__inline void operator delete[]( void* )
{}

__inline void operator delete[]( void*, IonMemory::BlockType )
{}

_STD_BEGIN

#   define MEMORY_DEF_NPALLOC( T ) \
        template<> class allocator<T>: public IonMemory::NoPtrAllocator<T> {};

    MEMORY_DEF_NPALLOC( char )
    MEMORY_DEF_NPALLOC( unsigned char )
    MEMORY_DEF_NPALLOC( short )
    MEMORY_DEF_NPALLOC( unsigned short )
    MEMORY_DEF_NPALLOC( int )
    MEMORY_DEF_NPALLOC( unsigned int )
    MEMORY_DEF_NPALLOC( long )
    MEMORY_DEF_NPALLOC( unsigned long )
    MEMORY_DEF_NPALLOC( bool )

_STD_END

#endif // Hf3faaf9f7b9c4e089d9e5b15c75358e7
```