Garbage Collection in the Reactive Deadline-Driven Environment of Timber

Martin Kero

Luleå University of Technology
MSc Programmes in Engineering
Department of Computer Science and Electrical Engineering
EISLAB
Garbage Collection in the Reactive Deadline-Driven Environment of Timber

Martin Kero

EISLAB
Dept. of Computer Science and Electrical Engineering
Luleå University of Technology
Luleå, Sweden

Supervisor:
Per Lindgren and Johan Nordlander
To Malin
Abstract

Software development for small, real-time and resource constrained, embedded systems is becoming increasingly complex. Multi-threading and object-orientation are examples of complicating factors. Even though automated memory management has shown significant advantages, it is commonly considered to be too costly and unpredictable for real-time systems.

Timber is a programming language that is based upon a reactive, deadline-driven, concurrent, and object-oriented programming model. It is furthermore based on a functional programming paradigm.

In this thesis we put forward a generic garbage collector for the run-time system of Timber. It is based upon a copying collector scheme and it especially makes use of the distinction between mutable and immutable data (induced by the language semantics), as well as the controlled accessibility of mutable data. Our collector is transparent, which means that it never preempts the application, and furthermore is incremental with extremely fine granularity. Minimal synchronization is needed between the application and the collector. Allocation takes an extremely small and constant amount of time (incrementing a pointer). Due to the reactive environment along with deadline-driven scheduling it will never compete for execution time with the application. Sufficient collector progress must thus be guaranteed by static analysis, and we address issues concerning this by briefly discussing memory usage behavior attributes (as a starting point for future work).
## CONTENTS

**Chapter 1 – Introduction**  
1.1 Background .............................................. 3  
1.2 Memory management in reactive deadline-driven environment .... 5  
1.3 Timber - A short introduction .......................... 6  
1.4 Thesis outline ............................................ 8  

**Chapter 2 – The Run-Time System of Timber**  
2.1 The features of the run-time system ....................... 9  
2.2 Data structures ............................................ 11  
2.3 Run-Time functionalities .................................. 12  
2.4 The idle state ............................................. 15  

**Chapter 3 – Garbage Collection**  
3.1 Problem definition ......................................... 17  
3.2 Troublesome issues ........................................ 18  
3.3 Copying Garbage Collection ................................ 19  
3.4 Scheduling garbage collection ............................ 25  

**Chapter 4 – Designing the Memory Manager for Timber**  
4.1 The book-keeper ............................................ 27  
4.2 The allocator ............................................... 28  
4.3 The collector ............................................... 29  
4.4 Ensuring robustness ....................................... 35  
4.5 Scheduling garbage collection ............................ 36  

**Chapter 5 – Conclusion**  
5.1 Related work .............................................. 40  
5.2 Future work ............................................... 40
The master’s degree project that corresponds to this master’s thesis has been performed at EISLAB, Luleå University of Technology, during the time period: September 2004 to May 2005. The project has been financed by the Research Trainee program at Luleå University of Technology.

First of all, I would like to thank my supervisors, Per Lindgren and Johan Nordlander, for invaluable technical expertise and for motivating me throughout the project, as well as for the extensive proof-reading of this thesis.

Second of all, I would like to thank my research trainee colleagues for an unforgettable year. Our joint office has been a stimulating environment, both academically and socially. The ”extra-curricular” activities has been very much fun, and hopefully we will find reasons to reunite the group in the future.

Last but not least, I would like to thank my family for their patience during the hectic periods of the project. They have been the source from where I have been able to recharge my batteries. Thank you for believing in me and in what I do!
Software development for small, real-time and resource constrained, embedded systems is becoming increasingly complex. Multi-threading and object-orientation are examples of complicating factors which already are employed in this field. Managing dynamic memory storage needs under such circumstances is a difficult task, and even though software systems with automatic memory management have shown many advantages over systems based on manual memory management, it is commonly considered to be too costly and unpredictable in a real-time context. The error-prone task of managing dynamic memory manually is often avoided by limiting the expressive power of the programming interface to only allow static (or scoped) memory usage. Nonetheless, automatic memory management is desirable when the tasks become more complex, and static/scoped approaches result in unwieldy systems.

Designing a hard real-time compliant automatic memory manager is a challenging task, and the purpose of this thesis is to show how the reactive deadline-driven run-time environment of the modern programming language Timber can make this task practically viable.

1.1 Background

In general, the memory management facility can be divided into two main parts: an allocator and a garbage collector (GC)\(^1\). The allocator serves the mutator (the application is often called the mutator due to the mutations it does to the heap \([1]\)), and the collector reclaims the storage space holding garbage data (i.e. the storage space that have been used by the mutator but no longer is). A memory manager in the context of hard real-time systems generally has requirements on small\(^2\) and strictly bounded delays to avoid

\(^1\)This division is an over-simplification, but for the purpose of describing the background of the real-time garbage collection problem, it is sufficient

\(^2\)Vague terms like small, good, efficient, etc. is used in this section to avoid too many details and technicalities that will be covered later on.
the risk of inducing violations of the mutator’s real-time constraints. In order to achieve this, several problematic issues can be identified.

**Scheduling Memory Management:** Scheduling concurrent processes in the context of a real-time system is a difficult task, and including scheduling of memory management work does not make it any easier. Ultimately, memory management work should never compete for execution time with the rest of the system, and at same time always keep the amount of free allocatable memory at a sufficient level.

**Fragmentation:** Allocations and deallocations of blocks of different sizes results in fragmentation. In the worst case, an allocation may fail even though the total amount of free memory is sufficient, because the allocator fails to find a contiguous memory area large enough to fit the request. The memory manager must in some way deal with the fragmentation problem in order to guarantee sufficient free memory contiguously.

**Bad Mutator Behavior:** Automatic memory management for arbitrary storage needs is an impossible task. There is always a possibility for a ”bad” mutator to defeat a specific memory manager [2]. Thus, the behavior of the mutator is an important property to take into account when designing the memory manager.

**Mutator-Collector Synchronization:** In order to guarantee small and bounded delays, the memory manager cannot occupy the CPU for extended periods of time. Due to this, it must allow the mutator to interleave its execution with a very fine granularity. Hence the memory manager must be incremental accomplishing its work in small and bounded increments. In order to guarantee heap consistency, it also needs to synchronize its view of the heap with the mutator in such way that it becomes aware, as well as makes the mutator aware, of possible heap mutations.

**Root-set scanning:** Many GC strategies are dependent upon a root-set when determine the aliveness of memory blocks on the heap, i.e. a block on the heap is alive if it is reachable from the root-set. The amount of work needed to identify the root-set is an important mutator property.

**Time/Space Overhead:** In the common case, designing a memory manager always comes down to the trade-off between time and space, i.e. execution time versus memory storage space. Real-time systems, especially in embedded contexts, often suffer from very scarce resources. Hence, the design decisions affecting this trade-off are important.

These problems are commonly accepted as matter of course for garbage collected systems. The gain of avoiding the error-prone task of managing memory is paid by extra overhead in terms of both space and time. Almost all real-time garbage collectors are based on a certain type of mutator, induced by an imperative programming language metaphor. This metaphor implies a typical mutator behavior which may very well be the actual reason to why some of mentioned problems arise, at least to some extents.
1.2 Memory management in reactive deadline-driven environment

In this thesis we present an implementation of a garbage collection technique that makes use of a reactive deadline-driven run-time system that distinguishes mutable and immutable data. Throughout the design we have striven to make the memory manager transparent to the rest of the system, and in particular we claim that:

1. The state of the memory manager does not affect the execution time of the mutator.

2. In the context of hard real-time constraints, memory management work never competes for execution time with the mutator.

3. Allocation time is bounded by a constant and can be made arbitrarily efficient [3].

4. Required memory overhead is bounded by the amount of simultaneously live memory plus eventual mutator need of new storage space during a collection cycle.

The support for the first two claims is inherited from the reactive deadline-driven environment in which the memory manager operates. The memory manager we present makes use of the characteristics that make this possible. The first claim is supported by a memory interface that always looks the same from the mutator perspective. The distinction between immutable and mutable data enables sparse book-keeping, which is the essence of transparency. Reactivity in combination with hard real-time constraints actually disallows the manager from competing for execution time. Through limited need for bookkeeping, the third claim is supported by the same transparency property as the first two. The fourth claim is supported by the actual garbage collection technique we have applied.

The main design objective of the memory manager has been, as mentioned before, to make it transparent. We have also stated that there is always a possibility for a "bad" mutator to defeat the manager, and each mutator characteristic that either enables or disables memory management are the key problems that need to be addressed. We have so far only addressed the characteristics that enable memory management and put forward how they support our transparency claims. The environment and its characteristics in which the manager will operate is actually the core of our design. We will later on address what "bad" mutator behavior is and how it may defeat the manager. We will also discuss how it first of all can be detected, and secondly, how it can be avoided. Automatic memory management in a hard real-time context cannot be successful in the presence of an arbitrarily "bad" mutator.

The remaining part of this chapter will present the programming language Timber, on which the reactive deadline-driven environment is based on. We will conclude with a short conspectus of the remaining chapters of this thesis.
1.3 Timber - A short introduction

During the last years, a new design methodology of embedded real-time systems has been developed. This methodology has rendered in a new programming language definition, called Timber[4]. The name is based on the words TIME, emBEdded, and Reactive. It is a reactive, real-time, concurrent, object-oriented, functional programming language, based upon O’Haskell[5, 6] which in turn is an extension to Haskell[7]. The development of the language is a joint effort by Luleå University of Technology, Chalmers University of Technology, and Oregon Health and Science University.

In brief, the language is based upon concurrently executing reactive objects [8]. The inter-object communication is message-based by means of synchronous and asynchronous message sends. A message send is equivalent to invoking a method of the recipient object.

Even though Timber is a general purpose language [9], it is primarily designed to target embedded systems, and we will discuss the aspects of the language in the context of embedded programming.

We will only describe the language Timber very briefly. It is still under development, and the syntax and semantics may be subject to refinements. Nonetheless, a thorough language description is forthcoming, and readers interested in learning more about the language should read [4, 9] or visit [10]. An example of a Timber program is shown in figure 1.1.

1.3.1 Records and Objects

Besides primitive data types such as integers, floating point numbers, etc., Timber includes user-defined records and primitive types to support object-orientation. Records can either be used to define regular data or to describe interfaces to objects.

Timber objects basically consist of two parts, an internal state and a communication interface. An object is instantiated by a template construct, which in turn defines the initial state of the object and its communication interface. The template construct can be seen as a module, offering an input interface and possibly requiring an output interface, when instantiated into an object.

The primitive object-oriented types are Action, Request, and Template, which all are subtypes of Cmd. The meaning of the Action and Request types are asynchronous and synchronous message sends, respectively. These actions and requests are collectively called methods. Template is the type defining the template command, from which objects are created.

A Timber program has to include a specific main template. The communication interface of this template is system dependent. For embedded devices the input interface usually contains a reset method and bindings from interrupts to actions. We will discuss this interface in a little more depth later on. The output interface defines the environment in which the Timber program will operate. In the context of embedded devices, it shall at least provide methods to read from and write to ports.

At system start up, the main template command will be executed, creating an instance of the object main and then executing the reset method. The system will supply the
1.3. Timber - A short introduction

```plaintext
1 sonar (port,alarm) =
2   template
3     t := baseline
4     ping = before (50*us) action
5       port.write(beepOn)
6     t := baseline
7     after (2*ms) stop
8     after (1*s) ping
9     stop = action
10    port.write(beepOff)
11   echo = before (5*ms) action
12     distance = k*(baseline-t)
13     if (distance < limit) then
14       alarm.on
15     return{
16       sonar = echo
17       start = ping
18     }
19 main regs =
20   template
21   s <- sonar ((regs!0xac00) a)
22   a <- alarm (regs!0xa3f0)
23   return {
24     reset = s.start
25     irqvector = [
26       (sonarIRQ, s.sonar),
27       (buttonIRQ, a.off)
28     ]
29   }

Figure 1.1: Example Timber program, A Sonar
```

main object with its environment.

### 1.3.2 Methods

A method is invoked by a message send command, either an asynchronous action or a synchronous request. A Timber program running on an embedded device can be seen as a set of concurrent objects, all awaiting external stimuli initially caused by interrupts. A method can basically do three things: it can update the state of the object, create new objects, and invoke methods of other objects. It is furthermore non-blocking, i.e. it cannot block the execution (e.g. indefinite loops). After an external stimulus, the chain of reactions will eventually fade out and the system will return to the state of waiting for new stimuli. We will refer to the time when the whole system is inactive and passive as the *idle state*, or *idle time*. 
Each message in Timber has a corresponding \textit{baseline} (earliest release time) and \textit{deadline} attached to it. By default, a message inherits baseline and deadline from its sender but for asynchronous messages both can be adjusted by \texttt{after} and \texttt{before} constructs.

\subsection*{1.3.3 Objects as concurrent reactive processes}

Each object in Timber has its own execution context, or thread of control. Inter-object communication is achieved by message-passing. Only one method within an object can be active at a time, and the object state is only accessible through its methods. This results in mutual exclusion of state mutations, usually referred to as state integrity. Furthermore, a method cannot block indefinitely, which leads to a controllable responsiveness of each object.

The input interface of the main template is, as mentioned earlier in our case, a reset method and an interrupt vector. The interrupt vector is a vector of pairs, connecting interrupt numbers to actions. This is how the environment obtains the ability to trigger reactions in a Timber program.

\subsection*{1.4 Thesis outline}

The remaining part of this thesis is divided into four chapters. Chapter 2 presents the run-time system of Timber, biased towards describing its mutator characteristics. In chapter 3, we survey the garbage collection technique that we have chosen to implement. Our specific contribution is presented in chapter 4, where we present our implementation, describing and discussing each significant design decision made. The thesis is then concluded in chapter 5, where we discuss related and future work.
In order to understand the run-time behavior of Timber, it is crucial to have some insight in the inner workings of its run-time system. In this chapter we address each functionality based on its semantic heritage, and describe how they together concretize the behavior, particularly in terms of memory usage.

The outline of this chapter is as follows: First of all a brief description of what the run-time system has to accomplish in order to substantiate the language semantics is given. Thereafter follows a more thorough description of each functional unit and their interactions.

### 2.1 The features of the run-time system

The run-time system of Timber and its functionalities is directly reflected by the semantics of the language. The following key features facilitating the semantics can be identified.

**Threading**: Facilitating the unique execution contexts for Timber objects.

**Scheduling**: The fundamental functionality of the run-time system to achieve concurrency between Timber objects, based on the baselines and deadlines of their methods.

**Message-passing**: Supplying sufficient infrastructure for inter-object communication.

**Time**: Ability to supply sufficient time information to make baselines and deadlines meaningful.

**Interrupt handling**: Functionality for receiving and distributing interrupts throughout the system.

**Environment interface**: Implementation of the interface to the environment.
Automatic memory management: Timber does not rely on explicit allocations and deallocations of dynamic data and needs an automatic memory manager to serve with garbage collection.

The semantics of Timber actually does not imply a specific scheduling algorithm. It rather states the following: *Every method invoked by a message send has to be finished within the specified time-line (between its baseline and its deadline).*

The language also basically suggests two levels of scheduling, one for messages (method invocations) within an object, and one between objects. The intra-object scheduling is non-preemptive in order to preserve state integrity, and priority inversion is solved by priority inheritance. The inter-object scheduling is preemptive, though, and realized in the run-time system by strict EDF (Earliest Deadline First), where the current deadline of an object is equivalent to the deadline of its most urgent message.

The message-passing mechanism in the run-time system is facilitated by message queues. Each object holds a queue of messages sorted by EDF. A message send will insert the message into the queue of the recipient object. The corresponding method will be executed when the object has that message first in queue and is scheduled to run. In addition to the local queues of each object, the run-time system also holds a queue of timed messages. Messages with a baseline ahead in time will be stored in the global queue of timed messages. This queue is sorted by earliest baseline first, and a timer is used to trigger the actual posting of these messages. In other words, the timer will be set to reflect the baseline of the first message in the queue (no need for repetitive system ticks).

To accomplish concurrency between objects, each object needs to have its own execution context; at least they need a reference to their current execution point in the code. We will further base the context on a non-shared stack environment, adding the current stack-pointer to each context. The ability to use a shared stack environment is under investigation, mostly based on the work by Baker presented in [11, 12]. Context switching and storage is accomplished by means of non-local goto, storing the code- and stack-pointers in jump-buffers attached to each object.

A notion of time is essential for the run-time system to accomplish correct scheduling. Baselines and deadlines are naturally expressed relatively to the actual occurrence of an event (external or timer interrupt). However, in terms of scheduling, these timing constraints are only possible to compare if they are in absolute form, i.e. $D_1$ is earlier than $D_2$ if $D_1 < D_2$. This is accomplished by calculating the absolute time of these constraints by adding the current time (i.e. the time of the event) to the relative time.

The main task of the interrupt handling mechanism is to serve as an interface between the interrupts and the message-passing mechanism. This is solved by a generic interrupt handler, which translates interrupts into messages.

The run-time system will allow context-switches to occur at three occasions. First of all, a context-switch may occur after a method is finished. This means that when the method finishes, the object will return control back to the scheduler, which in turn will do a possible context switch. The second case is after an interrupt has occurred and the
corresponding message is sent. Both the first and the second case may cause a context-switch, but will not necessarily do so. The third and final case (which will always cause a context-switch) occurs at a synchronous request. To be able to receive the requested value, the calling object has to let the recipient object execute the method, and thus a forced context-switch will occur.

The environment interface includes the low-level implementation of time, hardware initializations of interrupts, and environment methods offered to the main template. When a Timber program is in its idle state (all objects are inactive), the device is put into proper sleep-mode to lower power-consumption. The low-level implementation achieving this is also included in the environment interface.

The semantics of Timber is highly dependent on the ability to allocate memory storage dynamically. Furthermore, as mentioned, the language does not include any explicit allocation/deallocation commands. It is thus crucial to include a garbage collector in the run-time system.

2.2 Data structures

The primitive data structures present in the run-time system of Timber can be divided into three distinct groups; to wit; objects, messages, and regular data. The run-time characteristics and purpose of these structures differ radically from each other.

Objects An object is created when a template command is executed. The primitive object data structure consists of a deadline, a reference to its message queue, a reference to its state vector, and an execution context.

\[
\begin{array}{|c|}
\hline
\text{deadline} \\
\hline
\text{message queue} \\
\hline
\text{state} \\
\hline
\text{execution} \\
\hline
\text{context} \\
\hline
\end{array}
\]

Figure 2.1: The primitive object data structure.

The object structure is mostly immutable. The deadline of an object will be updated to reflect the deadline of its most urgent message but the field cannot explicitly be updated, only implicit due to a message send. The reference to the message queue is similar, where the actual reference will be updated implicitly. The execution context will of course be updated when ever the scheduler performs a context switch. However, the accessible field from the programming interface perspective, the reference to the state vector, is immutable. The vector contents is mutable, but
once instantiated, the reference to the state vector will never be updated; i.e. the object can never be assigned a new state vector.

Messages A message is created when a method is invoked. The primitive message data structure consists of a baseline, a deadline, a reference to the message closure, a reference to the recipient object, and a reference to the sender.

\begin{verbatim}
baseline
deadline
closure
to
from
\end{verbatim}

\textit{Figure 2.2: The primitive message data structure.}

Once a message is created it can never be updated. All fields in the message structure are immutable. The closure, which contains the reference to the executable method and eventual arguments, is also immutable. The \texttt{from} field, see figure 2.2, is used only for synchronous requests and is null for asynchronous messages.

Regular data Regular data can be of any type, but they are always immutable. The state vector of an object is treated (during allocation, etc.) as any regular data structure, even though it is mutable. As mentioned earlier the object itself protects it (state integrity).

2.2.1 Run-Time structure

The run-time structure will of course change over, time but a typical pattern can be discerned based on the two distinct states the run-time system can be in: either idling or active. Figure 2.3 illustrates an example of how the run-time structure might look when the system is in its idle state. In this simple example, the interrupt vector only consists of two messages, and one message is in the queue of timed messages (i.e. a baseline ahead in time).

During the idle state, no messages can reside in the message queues of the objects, and all run-time stacks are empty.

2.3 Run-Time functionalities

The run-time system supplies a Timber application with a set of functionalities to create and manipulate primitive data structures as well as their inter-relationships.
2.3. Run-Time functionalities

2.3.1 Message-related functionalities

Creating messages This functionality creates and posts a message, either asynchronous (action) or synchronous (request). A reference to the recipient object and the closure is given as parameters which will be stored in a freshly allocated message. A baseline and a deadline will be set properly, and then, based on the baseline of the message, the message will either be posted to the recipient object, or to the queue of timed messages. If the deadline is not explicitly given (by a before construct), the message inherits it from the sending object. If the deadline have not been given at all (for the whole chain of reactions), the message gets a deadline with the symbolic value 0, which is the earliest deadline a message can get. If the message is synchronous, which intrinsically cannot have a baseline ahead in time, it will be posted to the recipient object and then a context switch will occur. The requested value is stored in the sending object and returned when the scheduler switches back to the senders context.

Posting messages As mentioned, messages can either be posted to the recipient object, or to the queue of timed messages (baseline ahead in time). When a message is posted to the recipient object it will be inserted into its message queue, which is sorted by EDF. If the message is the object’s most urgent one, the deadline of the object will be updated. The object is then inserted into the ready queue (if the object is not busy, i.e. currently executing). Posting a message with a baseline ahead in time is very similar, but in this case the global queue of timed message is
the target, which is sorted by earliest baseline first, instead of EDF. If the message has the baseline that is closest in time (i.e. ends up first in the queue), the timer will be set to reflect the new baseline.

**Receiving messages** When an object is scheduled to run it will dequeue the first message in its message queue and execute the corresponding method. In addition to the eventual arguments stored in the closure, the reference to the state vector is sent as an argument to the method. When the method is finished executing, the message is dropped (in case of a request, the reply value is stored in the sending object first).

### 2.3.2 Object-related functionalities

**Creating objects** A reference to the state vector is given as a parameter to the create procedure. A new object is allocated as well as a new run-time stack (note that the possibility of using a shared stack is under investigation). The reference to the state vector is stored in the object and its execution context is initialized. A reference to the object is then returned.

**The event loop** Each object resides in an event loop. The procedure in this loop is as follows:

1. Dequeue the first message in its queue.
2. Execute the corresponding method with eventual arguments + reference to the state vector.
3. Store eventual reply value in the sending object.
4. Drop the message.
5. If the object has more messages in the queue, insert it into the ready queue.
6. Call the scheduler.
7. Go to 1.

**Insert object into the ready queue** The insert procedure either inserts the object into the ready queue or moves it towards the front if it already is in the queue. The queue is sorted by EDF, and since deadlines never increase, an object will never be moved towards the end of the queue. The procedure can be described as finding the correct position in the queue, insert the object there, and a possible reference to the object further behind in the queue is removed. If the object is encountered during the search for correct position, the insert procedure immediately returns because the object is already at the correct position in the queue (deadlines can never increase).

**Context switch (dispatch)** A context switch, or dispatch, is performed by non-local goto. The procedure simply stores the context of the current object, and loads the context of the object first in the ready queue.
2.3.3 Interrupt-related functionalities

The run-time system deals with two types of interrupts, external and timer interrupts.

**Timer interrupts** A timer is used to trigger timed messages to be posted. Timed messages are stored in a queue sorted by earliest baseline first. The timer is set to reflect the baseline of the message first in queue. When a timer interrupt occurs, all messages whose baseline has expired will be dequeued and posted to the recipient objects. If there is any messages left in the queue, the timer is reset to reflect the baseline of the first message in queue. The current object is inserted into the ready queue and the scheduler is called to perform a context switch.

**External interrupts** As aforementioned, bindings between interrupt numbers and actions (a message corresponding to an external interrupt can never be synchronous) are expressed as an interrupt vector. At the occurrence of an external interrupt, the correct message will be extracted from the interrupt vector and a copy of it will be sent to the recipient object. In cases when the deadline of the message is very short, and the risk of violating the state integrity of the recipient object is nonexistent; the method can be executed without posting a message (i.e. bypassing the scheduler).

2.4 The idle state

We have already mentioned that when the system is idle, i.e. no messages to process, the CPU is put into proper sleep mode. This is achieved by an idle object, which instead of residing in the event loop, resides in a sleep loop. It will furthermore always be the last object in the ready queue, due to its symbolic deadline INFINITYPLUS1 (the latest possible deadline for a message is INFINITY). All in all, it means that as soon as the system becomes idle, the idle object will be scheduled to run, but instead of processing any messages, it puts the device into sleep mode, which will reduce power consumption. This gives an intrinsic connection between the work-load and the overall power consumption of a Timber program.
Garbage collection (GC), or automatic memory management, is the process of detecting and reclaiming dead memory block of storage space and make it reusable. A block is dead if it is no longer reachable from the application, that is, the application has no references to the block. The term *aliveness* is defined by the following statement: *A memory block is dead and subject for storage reclamation if and only if the application no longer has any references to it.* In multi-threaded systems it is obvious that the aliveness of a block in shared memory space is a global attribute. In general, all dynamically allocated blocks in memory share the globality in their aliveness attribute. The general approach actually simplifies the definition of aliveness, including all dynamic memory blocks, not just those that are explicitly shared among more than one thread or process. This approach also concurs with the perspective of heap globality, and generalizes the work of the memory manager.

There exists a wide variety of techniques for performing garbage collection, but there is a significant distinction between two major families of collectors. On one hand we have reference counting collectors; that is, as it says, counting references to memory blocks. The garbage detection is achieved by detecting when the counter reaches the value of zero. On the other hand we have the vast majority of collectors, the so called tracing collectors. Instead of storing the aliveness information locally within each block, the tracing collector is based upon a *root set* that all other live blocks are reachable from. The collector traces the graph of references beginning in the root set of blocks to find all live blocks, then everything else on the heap is in fact garbage.

A reference counting garbage collector has been implemented for Timber, but not tested to any greater extent [13].

### 3.1 Problem definition

The GC has to fulfill two functionalities: it has to be able to detect garbage blocks in memory, and it has to be able to reclaim the storage area used by garbage blocks and
Garbage Collection

make them reusable [14].

In addition to the functional attributes we have a few non-functional dittos due to the real-time context [15].

**Predictability** This property has to be addressed both in terms of execution time and memory utilization.

**Schedulability** The scheduler of the run-time environment has to be able to schedule the tasks of the GC. This attribute closely determines the predictability.

**Efficiency** CPU time is expensive in small embedded systems. This property has to be addressed in terms of both time and memory space.

**Increment ability** The application cannot be halted by the GC in extended time periods. Thus, the collector has to allow itself to be interrupted and furthermore even allow the application to make changes to the heap in the middle of a garbage collection cycle.

Another property that is addressed in this context is *robustness* [15]. The meaning of robustness in a real-time perspective is somewhat unclear. In what sense does a real-time system have to be more robust than any other system with an equivalently critical task (except for timing)? On the other hand, systems that are put under real-time constraints are often critical in other aspects too. Throughout this thesis, the robustness property of a GC shall be understood as its ability to fulfill its main task - detect and reclaim - and the GC system is robust if it does not fail to accomplish that task.

Furthermore, all these properties has to be quantified in a measurable way for us to be able to make fair assessments of different approaches. However, a thorough investigation of this topic will not fit within the scope of this thesis.

### 3.2 Troublesome issues

Besides the conditions addressed in the problem definition, the task also comes with some additional well known issues that will throw spanners in the work of the GC. These issues will be addressed one by one, and some of them deserves some extra attention.

#### 3.2.1 Fragmentation

When blocks of different sizes are allocated and deallocated, the memory will be suffering from something called *fragmentation*. The problem occurs when small pieces of free memory are spread throughout the memory, forming an discontiguous free-space. Although the total amount of free memory might be sufficient, an allocation big enough may fail to fit within any of the smaller pieces.

The problem can be solved in two ways, at least. One way is to use one or more fixed sizes of blocks. Examples of different allocation policies using fixed block sizes are
segregated free lists [16] and buddy systems [17, 18]. Both of these policies are based upon the idea of splitting and coalescing memory blocks for a more accurate match between the block size and the actual requested storage space. The other way of solving this problem is to compact the heap by means of copying and moving live data so that free memory becomes contiguous.

The concept of fragmentation is commonly divided into two types: internal and external fragmentation [19]. Internal fragmentation is the result of allocating more memory than is requested, thus causing memory storage to be occupied but unused. It occurs when blocks of fixed sizes are used. The division is slightly misleading, due to the fact that internal fragmentation is the result of a preventative action taken to avoid external fragmentation. It would be more appropriate to describe internal fragmentation as an induced memory overhead due to defragmentation (thus giving the term defragmentation a more fundamental meaning, including both actual defragmentation as well as preventative actions).

Generally, designing the defragmentation process ultimately comes down to the trade-off between overhead in execution time and overhead in storage space.

### 3.2.2 Memory overhead

Striving for efficient GC and defragmentation in terms of time, often leads to unacceptable overhead in storage needs. Storage resources in embedded systems are often as scarce as CPU time, and sometimes even worse. In an ideal world, garbage collection and defragmentation would be extremely efficient, totally transparent and without overhead in memory usage. This is of course impossible in practice. The design should at least strive to optimize the process in such way that it would minimize the overhead, and find a trade-off level that is acceptable from both perspectives.

### 3.3 Copying Garbage Collection

A copying garbage collector combines the main task of a GC with defragmentation. Some researchers does not use the garbage collection term to describe this process - because
the collection happens implicitly, the garbage collection is actually a side-effect during defragmentation [14].

![Diagram of Garbage Collection Process]

Figure 3.2: Before and after a copying collection cycle

The process can be described as copying all live data into a contiguous memory area, the rest of the memory then being garbage and available for reuse. The first copying collector, presented in 1963, was the Minsky [20] collector for LISP. It used secondary storage, i.e. a file on disk, as target space for copying. Today this approach would not be very efficient, due to the several orders of magnitude slower file operations in comparison to memory operations. Fenichel’s and Yochelson’s [21] collector divides the heap into two semispaces where only one of them is active at the same time. A collection cycle is performed by copying all live blocks from the active space (fromspace) into the inactive space (tospace). Then the labels are switched, i.e. the previously active space becomes inactive and vice versa. To avoid copying the same block more than once, a forwarding pointer is installed in the old copy. Cheney [22] presented a non-recursive copying traversal algorithm using a scavenging scan pointer in contrast to the recursive approach of Fenichel’s and Yochelson’s collector. The basic idea is to scan the copied
blocks for references into fromspace, and if such a reference is encountered, copy it into tospace (see figure 3.3).

![Diagram of Cheney algorithm's breadth-first traversal]

Figure 3.3: The Cheney algorithm’s breadth-first traversal

### 3.3.1 Incremental copying collector

An incremental collector performs garbage collection in small and bounded increments. In contrast to "stop-and-copy" collectors, incremental collection is applicable in real-time systems.

Dijkstra et. al. [1] presents the *tricolor marking* abstraction. This is useful in understanding how garbage collection can be made incremental. During a garbage collection cycle, a block on the heap can be in three different states. It can be undetected (possibly garbage), detected (alive), or processed. The tricolor marking is based upon the idea of looking at the blocks with different colors. An undetected block is seen as white, detected as gray, and processed as black. At the beginning of a collection cycle, all blocks are white. When the collector traces the graph of references, it will color white blocks
gray and gray blocks black. The order in which this is done is dependent on the type of tracing collector that is used, but for an incremental copying collector, it is suitable to interleave the detection and copying in a *breadth-first* manner: before a gray block is colored black, all its white descendants (outgoing edges) are colored gray. This will lead to the phenomenon that is the essence of the *tricolor invariant*; to wit, a black block can never have an outgoing reference to a white block. There always has to be a gray block in between them. A violation of this invariant will cause a block that is alive to be reclaimed as garbage and possibly getting its content corrupted. This is not a problem for a stop-the-world collector, but when interrupts are allowed, the application may mutate the heap in the middle of a collection cycle and thereby cause a violation. To be able to avoid this kind of possible violation, the mutator and the collector needs to be synchronized - either by catching mutations or making it impossible to mutate the heap in such way that a violation may occur.

![Diagram of tricolor invariant](image)

*Figure 3.4: New allocation violating the tri-coloring invariant.*

Baker’s collector [23] is an adaptation of the Cheney collector, extended with real-time capabilities. It is incremental with a *tospace invariant*; that is, the mutator can only access blocks in tospace. This is accomplished by a *read barrier* that catches all heap reads, and if such a read is made in fromspace, the block is copied into tospace first (i.e. a ”copy-on-demand” policy). The intrinsic tricolor marking in Cheney’s collector is shown in figure 3.3, where everything behind the scan pointer is black and everything in front of it is gray. To avoid scanning blocks allocated during a collection cycle (i.e. allocating them as black) new allocations, are done at the other end of tospace. Brooks [24] extends Baker’s collector by always indrecting access to the block through the forwarding pointer, instead of checking if a forwarding pointer exists (i.e. if the block is already copied). If a block has not been copied, this pointer points to the block itself.

### 3.3.2 Generations

A significant drawback of a copying collector scheme is the possibility of copying long-lived blocks back and forth between the spaces when only short-lived blocks are collected. To remedy this problem one could use more than two spaces and separate the blocks between the spaces by their age. Due to the distinction in age, these spaces are commonly
referred to as *generations*. Examples of early copying collectors using generations are the Lieberman and Hewitt collector [25] and the Ungar collector [26]. The main difficulty in designing a copying collector based on generations, which was identified even in the early days, is the detection and treatment of inter-generation pointers. To avoid traversing a generation that is not subject for garbage collection, these pointers need to be stored as part of the root set. E.g., the root set for garbage collecting generation A has to include all pointers from generation B to generation A. If these pointers is not known a priori, generation B must be traversed in order to find them.

The age of a block can either be determined dynamically or statically. Lieberman and Hewitt realized that if the expected lifetime of a block is known when it is allocated, generational garbage collection would be much more efficient. The necessary bookkeeping in order to enable blocks to grow old, as well as the work needed to *promote* a block to an older generation, both carry significant overhead.

Engelstad and Vandendorp [27] present a real-time copying collector (based on Baker’s collector) using generations. Their performance claims are based on the observation that young blocks are more likely to die, and references seldom point from older to younger blocks. Hudson and Moss [28] focused on collection of older generations, which had been overseen by most previous work due to the much more volatile younger generations.

### 3.3.3 Coherence, consistency, and conservatism

In general, the coherence problem is basically due to multiple processes sharing mutable data. Typical coherence actions for garbage collected systems are taken to *protect* the collector from mutations made to the heap by the mutator. A copying collector generally suffers from an even worse coherence problem, as the mutator must also be protected from the mutations made by the collector. This situation is commonly referred to as a *multiple readers, multiple writers* problem. However, the needed protection in terms of induced inconsistencies is not that severe. The reason is that collector and mutator do not necessarily need to have a coherent and consistent view of the heap. For instance, the garbage collector may very well view some dead blocks as being alive (which only results in some unnecessarily occupied memory), as long as it does not view live blocks as being dead (which would have severe consequences). All garbage collectors are more or less conservative, in such way that some dead blocks may be treated as live because the cost of determining if it really is dead is too high. Ultimately, the requirement is that the tricolor invariant must not be broken, which means no live blocks should be collected. Another typical difference between collectors in terms of conservatism is how blocks allocated during a collection cycle should be treated. Baker’s collector is a typical example of a conservative collector due to the fact that it allocates blocks black, which means that all garbage produced during a collection cycle will go uncollected until next cycle.
### 3.3.4 Barrier methods

The needed synchronization between collector and mutator is generally solved by so-called *barriers*, catching heap reads and writes. As mentioned previously, the tricolor invariant must be upheld in order to avoid collecting non-garbage blocks. Generally, copying collectors are based upon a read barrier (e.g., Baker’s collector), which in essence is used to protect the mutator from seeing invalid pointers. Zorn [29] evaluates the performance of different read and write barrier implementations, which shows that read barriers are expensive on conventional machines mainly because reads occur very frequently.

### 3.3.5 Efficient allocation

Due to the fact that a copying collector defers collection work to a collection cycle instead of distributing management work among mutator work\(^1\), allocation can be made arbitrarily efficient. In essence, the only thing needed for allocating a block of size \(n\) is to adjust the *free pointer* by \(n\). Appel [3] claims that garbage collection can be made faster than stack allocation if sufficient physical memory is available. The main support for this claim is due to the independency between the performance of a copying collector and the amount of garbage produced (in contrast to stack allocation, where "garbage" must be popped off the stack).

### 3.3.6 Distinguishing mutable and immutable data

Doligez and Leroy [30], as well as Huelsbergen and Larus [31], identify the advantage of distinguishing mutable and immutable blocks from a garbage collectors perspective. This is especially an advantage for a copying collector, due to its consistency issues. Basically, in terms of consistency, mutable and immutable data differ in such way that access to two different copies of the same immutable data structure does not result in incoherence, whereas mutator access to multiple copies of a mutable data structure does.

### 3.3.7 Efficient scavenging

We have so far not discussed how descendants are detected in gray blocks. Although it might look like a trivial task to identify pointers, an misinterpreted field may cause, if not severe failures, at least overhead in terms of unnecessary work. Type information such as pointer locations is very useful in accomplishing robust and efficient scavenging.

### 3.3.8 Predictability and Efficiency

If we would be able to determine the amount of live data at any given point in time, the copying GC would be almost hundred percent predictable. The execution time of the collector is directly proportional to the amount of data that currently is alive.

---

\(^1\)Many copying collectors distribute work among mutator work during a collection cycle, but when the collector is inactive, no garbage collection work is done.
to the fact that the copying GC is actually a defragmentation process and GC is a side-effect, it would be unfair to evaluate its efficiency only in terms of collected garbage. The copying collector suffers from inefficiency when it performs unnecessary defragmentation. In the worst case, it would copy half the heap, accomplishing neither defragmentation nor garbage collection.

On the other hand, when the amount of live data is small and there is a lot of garbage produced, the collector would be very efficient. In the best case, it would collect half the heap of garbage almost without doing anything at all, at least not copying anything.

### 3.4 Scheduling garbage collection

Scheduling garbage collection for embedded real-time systems is not a trivial task. Many garbage collection techniques rely on explicit invocations; that is, the collector is not concurrent but rather triggered by memory operations such as pointer read/write and allocation. In order to guarantee small and strictly bounded delays, this approach is not acceptable due to its worst-case performance, where the small delays quickly add up into large ones. Henriksson [32, 15] surveys the problem of scheduling garbage collection in embedded systems with mixed high- and low-priority processes. The basic idea is to avoid garbage collection work during execution of high-priority processes, and trigger collector work as described above only when low-priority processes execute. However, the avoided (but necessary) collector work during high-priority processes is deferred to be done as soon as the high-priority process is finished executing, see figure 3.5.

![Figure 3.5: CPU time distribution according to Henriksson’s scheduling policy.](image)

Henriksson also relies on a copy-on-demand policy, where a write-barrier upholds the tri-color invariant by immediately copying white blocks when they are referenced. To avoid the delays of copying blocks on pointer assignments during execution of high-priority processes, a lazy evacuation scheme is used, where only the space needed in to-space is allocated, but the actual block is not copied.
3.4.1 Guaranteeing GC progress

In order to guarantee enough progress for the garbage collector, a sound metric needs to be identified that corresponds to the amount of work needed in order to supply the mutator with sufficient storage space. For a copying collector, it can be described as finishing a collection cycle (copy all live blocks into tospace) before the amount of free storage in tospace is consumed. The amount of work needed to finish a copying collection cycle is intrinsically dependent on the amount live memory that has to be copied. In addition to that, the memory in tospace is not only consumed by the copying process, but also by the mutator through new allocations.\(^2\) Henriksson describes a ratio between the amount of garbage collection work and new allocated blocks in tospace. This ratio must always be greater or equal to the ratio between the worst-case amount of garbage collection work needed for a whole cycle, and the minimum amount of available storage space in tospace for new allocations. If this can be upheld, fromspace can be guaranteed to be empty before tospace is filled up. Even though this is a sound way of determine the GC progress, the metric used is not easily derived. For instance, how should the amount of garbage collection work be measured? Henriksson describes a metric that can be derived in run-time as an approximation of collection work, and he addresses the problems of estimated metrics. However, even if it is possible (with more or less overhead) to measure and calculate the actual work needed as well as the amount of new allocation in run-time, this does still not solve the fundamental problem of guaranteeing sufficient GC progress, which needs accurate values in the worst-case scenario. Worst-case attributes and metrics for copying collectors are discussed in chapter 5.

\(^2\)We assume new allocations are done in tospace. If allocations are done in fromspace, then the amount of free space preserved in fromspace during a collection cycle must be sufficient, which ultimately is the same problem as we describe.
4.1 The book-keeper

The book-keeper is the unit that keeps track of all pointers, flags, and so forth, for the memory manager. Its only external control input will be at system initialization, when all initial values will be set. The system initialization is divided into two steps. The first step consists of initialization and preparation for static allocations, which simply means that all static allocations will be made sequentially in the free-space in memory. The second step is a preparation of the heap for dynamic allocations.

The information kept by the book-keeper is the addresses to the bottom and the top of the heap, the current free pointer, the current scan pointer, the address of the space border (between fromspace and tospace), and the current allocation direction. The information is initialized to the values shown in figure 4.1, where the scan and free pointers start at the bottom of the heap and the space border is set to the top of the heap. The allocation direction is initially forward. The dynamic behavior of this information will
be covered further on, but the common denominator for all accesses is atomicity. That is, any update made to this information is done as an atomic operation.

### 4.2 The allocator

Depending on which space that is currently active, the allocator will consume heap storage either upwards or downwards in the address space. In either case, an allocation is performed by simply incrementing/decrementing a pointer, which makes it extremely efficient. Due to Timber’s distinct groups of primitive data structures and strong static type safety, the mutator can store invaluable type information in each block. In addition to the size information, pointer locations within the block are known at the moment of allocation. The allocator only supplies storage space given a requested size, but the mutator requests storage space for an extra field which will hold a pointer to a statically initialized array holding the type information called `gcinfo`. An example is given in figure 4.2, where the first element in the array is the size, and then a null-terminated series of pointer offsets is stored. The first field in every data structure is the gcinfo, which will simplify the work of the collector substantially as we will see later on.

---

**Figure 4.1:** Pointers and information managed by the book-keeper.

**Figure 4.2:** Example of how type information is stored
4.3 The collector

The triggering factor for initializing a new collection cycle is a well debated issue, which we will cover later on in this chapter. However, in order to describe the procedures of the collector, it is important to understand the state of the system when a collection is initialized. Basically, the information we will utilize lies at the core of Timbers reactive deadline-driven behavior. First, a reactive system will always strive to reach the state of idling. Second, scheduling reactions to events is driven by hard deadlines. The question is when we can perform garbage collection. We will, without penetrating the subject any further in this section, assume that the collector does not have a deadline, and that it therefore only be restricted to execute only when the system is idle.

The procedure of the collector is to switch the space-labels (flip) and copy all live data from the previously active space (now fromspace) into the contiguous free, currently active, space (now tospace). The tracing scheme is based on the Cheney algorithm [22] with a scavenging scan pointer, which means that a collection cycle will be finished when the scan pointer catches up with the free pointer.

4.3.1 The flip

The immediate action performed by the collector when a new cycle is triggered is a so called flip. The purpose of the flip is to activate the free contiguous area of the heap, as well as inactivate the currently active one. This is performed by resetting a few pointers and flags (held by the book-keeper). In short, the procedure can be described as setting the space border to the current free pointer, moving the scan and free pointers either to the top or to the bottom of the heap (depending on the current allocation direction), and then reversing the allocation direction. An example of how the heap may look like before and after a flip is given in figure 4.3.

4.3.2 Copy root-set

The next step of the collector is to copy the root set into tospace. In order to accomplish this we need to identify the root set first, which in many cases include some form of scanning (possible heap references on the stack, in CPU-registers, and so forth). Fortunately, the root set in Timber during idle time is very easy to identify, and consists of only a couple of pointers (all stacks are empty during idle time). Basically, the root set can be derived from the channels through which the environment can trigger reactions within the Timber program. The channels are actually the interrupt vector and the queue of timed messages. The root set is thus only two pointers (the address of the interrupt vector and the address to the first message in the queue of timed messages).

Due to the possible mutation made to these root pointers, the copying process must be designed carefully. The easy way would be to copy the root-set in an atomic section, but this is not viable due to the delay induced in doing so. Our design decision concerning this can be described as atomically storing a copy of the root-pointers, then copying them. The new address is then stored atomically in case the original root-pointers have
not changed. In other words, we atomically retrieve and store the root-pointers, but the actual copying is done in between the atomic sections (see figure 4.4).

The basic condition that makes this approach work is that if the root pointer has changed (that means a new message has been enqueued), then it can only be to a newly allocated message in tospace, which need not to be copied. In case the first message gets dequeued (due to a timer interrupt), the second message in the queue will then become the first, which means that the root pointer changes but might very well point into fromspace. However, due to the fact that the dequeued message already has been copied, the scan pointer will find a reference to the previously second one in the linkage field (see figure 4.5).

The problem induced by mutability is much less severe for the interrupt vector, due to the absent linkage. The interrupt vector has a fixed size and is accessed by an index into the vector; thus the only possible mutation to this data structure is when substituting a message with a newly allocated message residing in tospace (i.e. the root cannot change to reference a message in fromspace in contrast to the timed queue).

### 4.3.3 The scavenging traversal

When the roots have been copied into tospace, the scan pointer will automatically be behind the free pointer. The scavenging traversal will extract the references in tospace
4.3. The Collector

Figure 4.4: Processing the mutable root pointers.

Figure 4.5: Example of how the process of copying the roots is correct, even though the first message in the timed queue is dequeued in the middle of the process. i) Original structure of the timed queue. ii) Structure of the queue immediately after copying A. iii) A is dequeued (semantically dead but still alive in tospace).
block by block and copy the descendants into tospace, thus moving the free pointer further ahead while chasing it. When the scan pointer catches up with free pointer it has intrinsically copied all live blocks into tospace, as well as scanned all references (both in copied blocks and in newly allocated) and ensured that no references exist from tospace into fromspace. When the scavenging process finds a reference, two different actions\(^1\) may take place depending on the value of the reference and the state of the referenced block. We will start by describing the most intrinsic action - copying the referenced block into tospace - and then describe the second action as well.

The copying mechanism will allocate storage space (in the same manner as any regular allocation) in tospace for the block. The needed size information is extracted from the gcinfo field stored as the first field of all blocks (see section 4.2). The actual copying phase will be described shortly, but for the sake of describing the different actions we will for now just assume that it will be copied into the newly allocated storage space. When the block has been copied, the address of the new copy will be stored in the gcinfo field, thus overwriting the type information in the old copy with a forwarding pointer. A check of this field as well as the actual value of the origin reference will determine the action that has to be taken. If the reference is into fromspace the gcinfo field is examined, otherwise no action is needed at all. Due to the fact that the type information is stored in a static array whose address is distinguishable from a reference into tospace, the proper action can be determined by comparing the value of the reference against the heap boundaries. If the gcinfo field is a forwarding pointer, then the value of it is stored as the new address of the origin reference. Thus, by simply examining the gcinfo field, avoidance of duplicated blocks in tospace is efficient and robust.

4.3.4 Copying objects

The mutable state vector in a Timber object has a special property, due to the concept of state integrity. Its fan-in is always one (only referenced by its parent object), which leads to an interesting phenomenon if treated properly. Barriers, used for the purpose of upholding the tri-coloring invariant are needed only when the mutator may gain access to duplicates of mutable data structures. We know that the only mutable data structure in Timber is the state vector.\(^2\) To be able to avoid all barriers we need to ensure that the mutator has access to only one copy of each state vector at any time.

The solution is to invert the regular copying order of the scheme when an object is reached. That is, we copy the state vector before the object, with the result of having both copies of the object referencing the same state vector, leaving no references to the old one for the mutator to gain access through. The actual copying of the object structure is then made as for any regular data structure.

---

\(^1\)Three actually, if the act of doing nothing counts as one.

\(^2\)This may seem like a contradicting statement considering the fact that fields in both the message and object structure are mutable. These fields are however only in use when the mutator is active, thus they can be ignored during idle time.
4.3.5 Copying the state vector

Allowing the mutator to interrupt the collector in the middle of copying a block is basically a requirement. The pause time that the collector may induce will otherwise be proportional to the maximum block size used by the application. The main problem of allowing interrupts in the middle of copying is that if the block gets mutated during that time, it may cause a possible loss of data. Fortunately, the only mutable data structure in Timber is the state vector, which we may treat differently during the phase of copying. We will introduce a dirty-flag that will be checked between each word-copy of the state.
To avoid barriers, the dirty-flag will be stored in the parent object, which will set it every time a method of that object is executed. If the dirty-flag is set in the middle of copying the state, the collector will start over and copy the state from the beginning again. Thus, the reference from the parent object to its state will be to the old copy until the whole state has been copied, and then it will be re-referenced into tospace. This introduces minimal overhead with no need of barriers, and preserves consistency locally. Furthermore, with some rather simple analysis methods that do not mutate the state can be identified, and in those cases the dirty-flag does not have to be set.

### 4.3.6 Copying messages and regular data

Due to the fact that all data structures other than the state vector are immutable, coherency and consistency of these blocks is not a problem. Even if the mutator has access to two copies of the same block, its semantics will not be affected, due to the fact that both copies will always be semantically identical. The only difference between two copies of the same block will be in the pointer fields. But the descendants will also always be immutable (with the exception for objects, but those data structures are treated differently as said before). It does not matter which copy of the descendant a block is referring to, they will anyway be identical. The conclusion is that duplicates of immutable blocks may co-exist without inducing any semantic changes, and furthermore, duplicates of any structure referencing immutable blocks may also co-exist without problems.

![Figure 4.7: Duplications of immutable blocks causes no consistency problems.](image)
4.4 Ensuring robustness

In order to guarantee that the collector never induces any inconsistencies, all possible scenarios must be taken into account and the collector algorithm must be verified to be correct for all those scenarios. Ultimately, at the end of a collection cycle, all live blocks must have been copied and no live references into fromspace may exist. We have so far argued that the distinction of immutable and mutable data structures makes the work of achieving consistency much more efficient. However, a set of possible scenarios (may be seen as so called worst case scenarios) can be identified.

4.4.1 Mutations behind the scan pointer

We have addressed the importance of allocating blocks as gray due to the possible references into fromspace they may contain. This can (as discussed in chapter 3) also be guarded by barrier methods, but in our case these costly barriers may be avoided altogether. What we have not addressed is the risk of mutations made behind the scan pointer (i.e. mutations in black blocks). These mutations may very well include pointer assignments, which refer to blocks in fromspace (an example is given in figure 4.8).

![Figure 4.8: Introducing a fromspace reference into a black block. During a reaction it may happen that a reference to a block in fromspace is retrieved from a block in front of the scan pointer (gray), and stored in a block behind the scan pointer (black).](image)

It may seem like this problem is the great pitfall in our quest to avoid barrier methods. However, due to the fact that this problem is a special case which does not involve any extra copying (in contrast to barriers used in a copy-on-demand policy), it can be made rather efficient. What we actually need to do is to scan the state vector that has been mutated and ensure that it does not have any fromspace references. The question is; how do we inform the scavenger about these malign mutations? Considering the scanning process, it would be convenient to inform the scavenger by allocating a special block (indicated by its gcinfo field) that this has occurred. What we need to store in this block (in addition to informing that it has occurred) is where the infected state vector is. When the scavenger reaches this write barrier, the scan pointer can be temporarily
moved backwards to the infected state vector, scanned for fromspace references, and then returned to its actual position. This solution is efficient in the sense that the mutator only needs to allocate a write barrier (allocations are very efficient) and store a reference to the state vector in it. The extra work (scanning etc.) needed is done solely by the collector.

![Image of a write barrier](image)

*Figure 4.9: Example of a write barrier.*

### 4.4.2 Duplicated objects during a reaction

Storing deadline information in an object might cause problems when it comes to scheduling. If duplicates of an object exist during a chain of reactions then there is a possibility that the deadline is updated in one copy, but the scheduler retrieves it from the other one. The easiest way to avoid this problem is to schedule messages instead of objects, thus removing the deadline information from the object. In addition, this would remove the need for local message queues. Due to the fact that this optimization is forthcoming and it will solve these problems, no effort will therefore be spent on solving them.

### 4.5 Scheduling garbage collection

In contrast to the system described by Henriksson [15], our system consists solely of "high-priority" processes. The reason is that every reaction will sooner or later reach its deadline, and the sooner the deadline is from the current point in time, the higher the priority is of that reaction. In a deadline-driven scheduling environment, the scheduling information is highly dynamic, which makes the integration of garbage collection a non-intrinsic task. In an abstract way, one might ask what the garbage collector should react to, as well as what timing constraints that reaction should have. The timing requirement of the GC is intrinsically described as something very different compared to the constraints of a reactive deadline-driven process. However, one could say that the garbage collector reacts to some sort of indicator of forthcoming memory shortage, and the deadline of the process is derived from the time it takes to exhaust the memory (i.e.}
the collector must finish the cycle before memory is exhausted). All in all, scheduling
garbage collection in our reactive deadline-driven system comes down to two questions.

1. When should a garbage collection cycle be triggered?

2. Under what condition can the garbage collector compete for execution time with
   the mutator processes? I.e. does the garbage collector have a deadline?

### 4.5.1 Triggering a garbage collection cycle

The triggering factor for initiating a garbage collection cycle can either be work-based
or time-based. The work-based trigger would in our case be some sort of threshold for
the amount of free space. That is, when the amount of free space drops below a certain
threshold, a garbage collection cycle is initiated. Time-based means that the collector
would be triggered with a certain periodicity, which in our case could be accomplished
by defining the garbage collector as a periodic process.

As long as memory never gets exhausted, both strategies are viable. It may very well
be so that for some applications mutator behavior one of the triggering factors is more
suitable, and for other cases the other one. In order to assess the two strategies versus
each other under different circumstances (mutator behavior, heap size, etc.), thorough
analysis and verification is needed. More about this in chapter 5.

### 4.5.2 Does the garbage collector need a deadline?

In order to compete for execution time with mutator processes, the garbage collector
would need a deadline. However, if the garbage collector had a deadline that at some
point would be earlier than the deadline for a mutator process, and the consequence
would be that the mutator process missed its deadline, what would the consequences be
in an overall system perspective? The question is ultimately which is worse: missing a
deadline or running out of memory? Our answer is they are equally bad. Due to the
fact that each reaction in a Timber system has a hard deadline, missing such a deadline
would have severe consequences, and therefore it is equally bad as a system crash due
to exhausted memory resources. In addition, a missed deadline is an indication of a
work-load that exceeds one hundred percent, which in a system scheduled by EDF can
result in a so called domino effect. That is, if one process misses its deadline it is very
likely that subsequent processes will miss their deadlines too, and in the worst case the
result would be that all subsequent processes would miss their deadlines (as long as the
work-load does not drop below one hundred percent).

So we argue that, instead of competing for execution time with the mutator processes,
the garbage collector will run as the idle object. That means that its deadline would be
the symbolic value INFINITYPLUS1. That in turn means that it will only execute when
no reactions are taking place. However, as we have discussed, the collector may very well
be preempted by mutator processes, and therefore the collector must be incremental.
Nonetheless, every small increment of garbage collection work is done during idle time. This is the actual reason to the fast identification and scanning of the root-set.

But, how do we know if there is enough idle time for us to be able to guarantee that the collector will make sufficient progress? This is a very complex subject which only will be addressed very briefly in chapter 5.
**Conclusion**

We have in this thesis put forward a design of a copying garbage collector crafted to suit the run-time environment and behavior of the Timber language. Although copying collectors suffer from a few severe drawbacks in the context of a resource constrained embedded systems, the language semantics of Timber facilitates many advantages, especially in terms of possible relaxations of consistency issues. We have presented an almost barrier-free collector, where the barrier (1) is on method level instead of single pointer assignments, and (2) the needed work is extremely small (allocating a block holding a reference to the state of the current object) and in particular the barrier always takes a constant amount of time. The collector is scheduled as the idle process, which means that it never competes for execution time with mutator processes. This, along with its transparency (efficient barrier), guarantees that the collector never induces any deadline violations of other processes. The four claims stated in chapter 1 have been verified to hold by:

1. The barrier needed is on method level and takes only a small constant amount of time. Due to the fact that this is the only needed synchronization, the collector is transparent to the rest of the system.

2. By scheduling the collector as the idle process, it will never compete for execution time with mutator processes.

3. We have showed that allocation takes a very small and constant amount of time.

4. The last claim is supported by the fact that a copying collector only needs a contiguous free space big enough to fit all currently live blocks in memory. Due to the incremental property, tospace must also be big enough for eventual new allocations.

Discussions about attributes that distinguish different behaviors of mutators have only been covered briefly (e.g. the discussion about what a "bad" mutator behavior is). This will, however, be further addressed in section 5.2 where we discuss future work.
5.1 Related work

Real-time garbage collection has been around for a long time but the requirements that formally makes a garbage collector "real-time"-compliant is rather weak. Statements like "...small and bounded delays..." are of course correct but rather vague. Even though most (probably all) work that relates to ours differs in both the approach (applied technique) as well as in terms of real-time requirements, many interesting publications have been made and we will cover a few of them (based on their relationship to our work).

Bacon et al. [33] describes a mostly non-copying garbage collector using segregated free lists (memory is divided into fixed-size pages, and each page is divided into blocks of a particular size). Defragmentation is done by copying objects from a fragmented page into another page. Relocation of object references is achieved by forwarding pointers, which are checked by a read barrier. Non-copying garbage collection is achieved by an incremental mark-sweep algorithm similar to that of Yuasa [34]. To bound the work of copying large data structures, large arrays are broken into fixed-size arraylets. The collector is not concurrent; i.e. the interleaving between mutator and collector is explicitly controlled. In comparison with our collector (which in retrospect is rather minimalistic), Bacon et al. present a more heavy-weighted collector probably suitable for larger systems.

Siebert [35] presents work on how to determine sufficient progress of the collector, which we only have covered very briefly. However, a discussion about the future work concerning this can be found in section 5.2.

In addition to the work that is presented by Henriksson [15], Kim et al. [36] give a scheduling policy for garbage collection which takes a rather different approach. Their policy schedules the incremental collector as the highest priority process, which in turn leads to reduced memory requirements due to minimized worst-case response time of the garbage collector. This may seem like an awkward approach in a real-time system, where time often is the most constrained resource. However, the evaluation shows a significant reduction in the worst-case memory requirement compared with, as they call it, a background policy. Based on a modified copying collector originally presented in [36], Kim et al. [37] also present a technique for estimating and bounding worst-case collection time.

For a discussion on the problematic issues of root scanning (which in our case always takes a constant amount of time) we refer the reader to the work by Siebert [38].

5.2 Future work

Due to the fact that this thesis only presents design work and intuitive assessments based on theoretical studies, much more work can be done in terms of evaluating our collector. In addition to evaluations, we will in future work focus on how static program analysis can be accomplished in order to make fair assessments on the worst-case memory usage of applications in relation to available idle time. We will furthermore (which is in line with the whole systematic approach of Timber) strive to integrate the garbage collector as early as possible in the process translating Timber source code into executable machine
code. That means, that we will define our memory management policy as a part of the language semantics, which in turn means that the most efficient garbage collector for a specific application may be considered. What we have done in this thesis is to build the fundamental generic part of a memory management infrastructure, that may (and will) be crafted to suit different applications. In order to accomplish this, we will strive for early integration of analysis in the compilation process, to determine memory usage behavior as well as required idle time for garbage collection.

The current run-time system is basically a prototype, and its purpose is, similar to the memory manager, to be a generic infrastructure. It may, again similar to the memory manager, be crafted on basis of the special needs and attributes of the application under consideration. However, the run-time system is still much of a prototype, and many optimizations are possible, e.g. unifying the message queue facility, schedule messages instead of objects, enabling the use of a shared stack environment, based on SRP (Stack Resource Policy) [11, 12], etc.

If these visions are realized, Timber will certainly become a fully-fledged low-weight real-time design/programming metaphor for tiny embedded devices.


REFERENCES


