Flexible automatic memory management for real-time and embedded systems

Flexible automatic memory management for real-time and embedded systems

Sven Gestegård Robertz

Licenciate Thesis, 2003

Department of Computer Science
Lund Institute of Technology
Lund University

ISSN 1404-1219
Dissertation 18, 2003
LU-CS-LIC:2003-2

Department of Computer Science
Lund Institute of Technology
Lund University
Box 118
SE-221 00 Lund
Sweden

Email: sven@cs.lth.se
WWW: http://www.cs.lth.se/~sven

Typeset using $\LaTeX 2_\varepsilon$

Printed in Sweden by KFS AB, Lund, 2003

© 2003 by Sven Gestegård Robertz

Abstract

The advent of safe languages like Java on the real-time systems scene motivates further research on efficient strategies for non-intrusive garbage collection and especially GC scheduling. This thesis presents new approaches to flexible and robust memory management from an engineering perspective and is a step towards *write once — run anywhere* with hard real-time performance.

The traditional approach to incremental GC scheduling, to perform garbage collection work in proportion to the amount of allocated memory, has drawbacks and in order to remedy this, a scheduling strategy, *time-triggered GC*, based on assigning a deadline for when the GC must complete its current cycle is proposed. It is shown that this strategy can give real-time performance that is equal to, or better than, that of an allocation-triggered GC. It is also shown that by using a deadline-based scheduler, the GC scheduling and, consequently, the real-time performance, is independent of a complex and error-prone work metric.

Time-triggered GC also allows a more high-level view on GC scheduling as the GC cycle level rather than on each individual increment is considered. This makes it possible to schedule GC as any other thread. It also makes the time-triggered strategy well suited for auto-tuning and it is shown how an adaptive GC scheduler can be implemented.

A novel approach of applying priorities to memory allocation is introduced and it is shown how this can be used to enhance the robustness of real-time applications. The proposed mechanisms can also be used to increase performance of systems with automatic memory management by limiting the amount of garbage collection work.

The ideas brought forward in this thesis have been implemented and validated in an experimental real-time Java environment.

Acknowledgments

I wish to thank Boris Magnusson, my supervisor and the leader of the Software Development Environments Group at the Department of Computer Science for his support. I am also most grateful to my assistant supervisors; Roger Henriksson, who introduced me to the field of real-time garbage collection, and Klas Nilsson, who has provided the automatic control and automation perspective. Thank you for your patient help, valuable input, and encouragement throughout this work. Görel Hedin was my supervisor in a previous project, at the beginning of my graduate studies, and taught me much about research and technical writing.

This work was done within the research project “Integrated Control and Scheduling” for which Karl-Erik Årzén, Klas Nilsson, and Ola Dahl wrote the original proposal.

The prototype implementations have been made in cooperation with other projects and much of the experimental work had not been possible without the assistance of others. I would like to thank Anders Ive for his help with implementing some of these ideas in the IVM virtual machine, Anders Nilsson for help with the Java to C compiler, Anton Cervin for valuable discussions on scheduling and control systems and for his implementation of CBS in the STORK kernel, and Anders Blomdell for helping me with the STORK/PowerPC platform.

Thanks to Torbjörn Ekman and Patrik Persson for interesting and enjoyable discussions on real-time systems development, to Ulf Asklund for teaching me a great deal about configuration management, and to Christian Andersson for assisting with LaTeX tips and tricks.

I am also thankful to Anne-Marie Westerberg, Lena Ohlsson, Peter Möller, Lars Nilsson, Tomas Richter and Jakob Westerberg for all help with practical details — everything had been much harder without you.

Finally, I thank everybody at the Department of Computer Science and LUCAS (Lund Center for Applied Software Research) for providing ideas, perspective, discussion, and good company.

This research project has been a collaboration between the Department of Automatic Control and the Department of Computer Science at Lund Institute of Technology. The work has been financially supported by ARTES (A network for Real-Time research and graduate Education in Sweden) and SSF (the Swedish Foundation for Strategic Research). The experiments have been carried out in cooperation with projects financed by VINNOVA (the Swedish Agency for Innovation Systems).

CONTENTS

1 Introduction **1**
1.1 Problem statement 2
1.2 About the thesis 5

2 Preliminaries **7**
2.1 Real-time systems 7
2.1.1 Timing requirements 7
2.1.2 Predictability 9
2.2 Scheduling 9
2.2.1 Fixed priority scheduling 10
2.2.2 Earliest deadline first scheduling 11
2.2.3 Co-existence of hard and soft processes 11
2.3 Memory management 13
2.3.1 Garbage collection 14
2.3.2 Incremental and real-time GC 15

3 Time-triggered garbage collection **19**
3.1 GC cycle time calculation 21
3.2 Using time as the GC work metric 26
3.3 Scheduling 28
3.3.1 Fixed priority scheduling 30
3.3.2 EDF scheduling 30
3.4 Summary 31

4 Adaptive garbage collection scheduling **33**
4.1 Introduction 33
4.2 Overview of an adaptive GC scheduler 35

4.3 Automatic GC cycle time estimation 37
4.4 GC workload estimation 40
4.4.1 Estimating accumulated GC workload 41
4.4.2 Estimating total GC work 44
4.5 Summary 47

5 Priorities for memory allocation 49
5.1 Introduction 49
5.2 Applying priorities to memory allocations 50
5.2.1 Avoiding out-of-memory situations 50
5.2.2 Improving performance by reducing GC work . . . 51
5.3 Non-critical allocations 52
5.3.1 Non-critical allocation limit 53
5.3.2 Fixed GC cycle length 53
5.4 Detailed description 55
5.4.1 Calculating the GC cycle length 55
5.4.2 Live memory and floating garbage 56
5.4.3 GC for the low priority processes 56
5.4.4 Non-critical limit calculations in the real world. . . 57
5.4.5 Time-based GC scheduling 59
5.4.6 Example 59
5.5 Non-critical memory in Java 61
5.6 Summary 62

6 Experimental verification 65
6.1 Time-triggered GC 67
6.2 GC cycle time auto-tuning 71
6.3 Priorities for memory allocation 72
6.3.1 Avoiding out-of-memory situations 72
6.3.2 Improving performance 75

7 Future work 77
7.1 Adaptive GC scheduling 77
7.2 Priorities for memory allocation 78
7.2.1 Configurable behaviour 78
7.2.2 Non-critical memory using aspects 79
7.3 GC scheduling interface 79
7.4 Feedback scheduling and QoS 80
7.5 Distributed hard real-time systems 80

8 Related Work 83
8.1 Time-based garbage collection scheduling 83
8.2 Adaptive GC scheduling 85
8.3 Memory Management in Real-Time Java 85
8.4 Soft references 87
8.5 Worst case and schedulability analysis 87

9 Conclusions 89
9.1 Contributions 90
9.2 Reflections 91

Bibliography 93

CHAPTER 1

INTRODUCTION

Memory management in real-time and embedded systems is handled in a very conservative manner. For reasons of safety and predictability, static memory management is often the technology of choice, but as embedded systems grow in complexity, dynamic memory management becomes increasingly desirable. Although more flexible than static memory management, manually managed dynamic memory tends to inflict new problems of predictability, robustness, and maintainability — important properties of embedded systems. Many of these problems can be overcome by automatic memory management, or *garbage collection (GC)*. With the advent of type-safe languages like Java on the real-time systems scene it becomes increasingly important to develop reliable, predictable and non-intrusive garbage collectors which are capable of meeting the memory allocation demands of our applications at all times. The garbage collector should also be transparent to the application developer and not require cumbersome manual tuning to be effective on any particular platform. This thesis proposes a new approach to garbage collection scheduling aimed at meeting these demands.

The focus of this thesis is on GC scheduling rather than algorithm design. Using either knowledge of the worst-case allocation need of the application, or by using auto-tuning techniques, it is possible to calculate a deadline for when garbage collection must be completed and new memory made available for allocation. Having an explicit deadline for the GC cycle implies that it would be possible to schedule GC using standard scheduling techniques, such as rate monotonic or earliest deadline first scheduling. This thesis investigates the feasibility of such an approach. Since the elapsed time determines when to run the garbage collector, we call the approach *time-triggered garbage collection*.

Another area of growing research interest and recent development is that of handling non-determinism in real-time systems, and an approach that has been successful is feedback scheduling. By using feedback control, the period times of the processes are dynamically altered in order to keep the total CPU utilization at a safe level. This is particularly useful in control systems, where it is the resulting control performance, rather than real-time performance, that is the ultimate goal. By getting the process scheduler into the loop, this allows co-design of control and real-time systems. Furthermore, worst-case analysis is not always feasible, due to non-determinism in modern computers, lack of engineering resources or simply that a design based on worst-case assumptions would be too pessimistic and therefore yield too low average resource utilization to be economically feasible. For these reasons, it is interesting to study adaptive memory management. This thesis presents two approaches aimed at enhancing the robustness of memory management for systems run in an unknown or changing environment.

1.1 Problem statement

This work comes from a practical engineering perspective and is aimed towards developing techniques that facilitate the production of embedded and real-time systems without the need for rigorous analysis and huge engineering effort that is currently required to develop hard real-time systems. This thesis addresses two categories of problems: The first is adding flexibility to embedded systems without jeopardizing their real-time properties. The second is how to implement hard real-time garbage collection in an actual run-time system.

Adding flexibility to hard real-time systems

In this thesis, the focus is on memory management. The reason is that the previous research on flexible real-time systems has focused on process scheduling and little attention has been given to memory management issues and their impact on process scheduling. Also, while many of the problems are generic to all kinds of resource allocation, memory allocation differs from CPU allocation in a major way in that preemption is not possible[1]. Therefore, running out of memory is likely to cause the entire system to fail while requesting too high CPU utilization may

[1] In systems with virtual memory, swapping and paging may be viewed as memory preemption, but this is uncommon in embedded systems as they typically lack secondary storage.

cause some or all processes to miss deadlines but the system may be able to continue executing with decreased performance.

Let us start by making three observations on real-time and embedded systems: The first one is that *the need for flexibility in hard real-time systems is increasing.* Component based software development helps facilitate code reuse and makes it possible to build systems quickly by composing and configuring components. While it is possible, in theory, to perform worst case and schedulability analysis on each configuration, constraints on the amount of available engineering resources may prohibit such analysis. Therefore, adaptive techniques like feedback scheduling are increasing in popularity as they allow a system to adapt its resource utilization in order to keep the system from overload while still producing an acceptable quality of service.

Another technique that is gaining interest is dynamic reconfiguration and code exchange where communicating devices may send pieces of code to each other in order to perform some cooperative task. In such a system, an introduction of a new device may cause pieces of code that were not part of the original design to be executed on other devices. This is facilitated on the programming language level by e.g., dynamic loading of code, but the run-time system aspects need further studies. For instance, in an environment where code is dynamically loaded and replaced at run-time, static worst-case analysis (and scheduling based thereupon) is not possible. Yet, it is desirable to include such techniques in hard real-time systems.

The second observation is that *not all hard real-time systems are safety critical.* A system is a hard real-time system if it fails or suffers major performance degradation if deadlines are missed. But, for some systems, that may be acceptable if the probability is low enough. This is also motivated by the high cost of the engineering effort required to make absolute guarantees that a system will never fail.

The final observation is that a problem with the current methods for real-time systems development is *the gap between theory and practice*; the real-time theory requires hard worst case calculations in order to guarantee schedulability. However, it is very common to use measurements or "gut feeling" estimates rather than exact analysis to obtain the worst case memory and CPU requirements and then, the quality of the real-time guarantees is no better than that of the worst case estimates. For those reasons, it may be better, both in terms of development costs and run-time performance, to reserve the hard, a priori analysis based methods for the development of systems which are safety critical and to use adaptive techniques for systems which are not.

Motivated by these observations, the high level goal of this work is to develop techniques for implementing hard real-time run-time systems, particularly memory managers, that are independent of a priori analysis of the application. That is, *if* an application is schedulable, the run-time system should be able to guarantee real-time performance — *write once, run anywhere* for hard real-time systems.

Making hard real-time memory management feasible in practice

The second problem addressed in this work is that previous research on hard real-time garbage collection may not be directly applicable when implementing actual real-time systems.

Firstly, one problem is the metric used to measure garbage collection work. A good metric is essential to both schedulability analysis and for the actual scheduling at run-time. Unfortunately, in much of the existing literature, the problem is either neglected or the reasoning is done on a too abstract level to be practically applicable.

Secondly, non-intrusiveness is a fundamental requirement on a hard real-time garbage collector as GC work must not cause processes to miss their deadlines. However, the common way of implementing real-time GC is to use an incremental garbage collector that performs small portions of work at each memory allocation — in line with the application processes — and previous research has often been content with showing that it is possible to find tight upper bounds on the lengths of each increment. That is not a good strategy if one wants to minimize latency and jitter due to garbage collection; even though each increment has a small upper bound, if a process makes many allocations the total delay caused by garbage collection will be large. Therefore, it is not enough to prove predictability — in actual product development it is equally important to have a scheduling model that allows maximum utilization of available resources.

Finally, previous real-time garbage collectors have required very fine grained analysis in order to tune them to a particular application, and the run-time scheduling has been done at the individual increment level. This has made the utilization of real-time GC difficult and tedious and the whole concept of automatic memory management in real-time systems has often been shunned.

This work is an attempt to provide a conceptual framework and techniques that are independent of the GC implementation and allow reasoning about garbage collection scheduling at a higher level, without abstracting away the difficulties. The goal is to make it possible to schedule garbage collection as any other task.

1.2 About the thesis

Outline

The rest of the thesis is organized as follows:

Chapter 2: Preliminaries describes the fundamental concepts of the areas of real-time computing and memory management and presents previous research on which this thesis is based.

Chapter 3: Time-triggered garbage collection introduces the idea of time-triggered garbage collection and discusses its impact in fixed-priority and earliest deadline first scheduled systems.

Chapter 4: Adaptive garbage collection scheduling discusses how a time-triggered garbage collector can be made auto-tuning and presents techniques for estimating the GC cycle length and the amount of work required to perform a GC cycle.

Chapter 5: Priorities for memory allocation presents a novel notion of applying priorities to memory allocations and shows how that can increase robustness and performance of real-time systems.

Chapter 6: Experimental verification presents experimental support for the proposed techniques.

Chapter 7: Future work outlines our plans for future research and points out possible areas of application for these ideas.

Chapter 8: Related work relates the work presented in this thesis with previous results in the areas of garbage collection scheduling, memory management for real-time Java and worst case analysis.

Chapter 9: Conclusions summarizes the contributions of this thesis.

Publications

This thesis is largely based on published papers. The exception is the GC work estimation presented in Chapter 4 which is work in progress and has not yet been submitted for publication.

Chapter 3 and part of Chapter 4 are based on the paper

> Sven Gestegård Robertz and Roger Henriksson, *Time-Triggered Garbage Collection — Robust and Adaptive Real-Time GC Scheduling for Embedded Systems*, which will appear in Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems – 2003 (LCTES'03) [45].

Chapter 5 and the corresponding experiments was published as

> Sven Gestegård Robertz, *Applying Priorities to Memory Allocation* in Proceedings of the 2002 International Symposium on Memory Management (ISMM'02) [44].

The prototype implementations used in the experimental verification are closely related to the development of the *garbage collector interface* which was presented in

> Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson and Sven Gestegård Robertz, *Garbage Collector Interface*, in Proceedings of NWPER'02 [24].

CHAPTER 2

PRELIMINARIES

This chapter briefly presents the fundamental concepts of real-time systems, scheduling and memory management. Previous research in the fields of scheduling and automatic memory management for real-time systems, which forms a base for the remainder of this thesis, is presented and discussed.

2.1 Real-time systems

The task of a computer program is to produce some output based on its input values. The fundamental definition of correctness is, of course, that the program produces the right output for any valid input values but for some systems, typically those that interact with an external environment in some way, this is not enough. In addition to producing the right output, the definition of correctness is strengthened to also require that such a system produces the output before a given time, the *deadline*. Such systems are called *real-time systems* and typical examples are found in the areas of automatic control, communications, audio/video, interactive computer programs, etc.

2.1.1 Timing requirements

The term real-time systems represent a wide range of applications with widely varying timing requirements, and the consequences of failing to meet deadlines also range from minor inconveniences to total failures. Computer systems can be categorized based on their real-time requirements and a brief overview of the taxonomy is given here.

Batch systems

Most computer programs do not have any real-time requirements other than that it, naturally, is desirable that the result is produced as quickly as possible in order to make the program practically usable. Examples of such programs are compilers, mathematical programs, etc. Such programs are called batch systems, as they typically take a batch of input, perform some processing, and output the result. In batch systems, the correctness of the system is completely independent of the time it takes to produce the output.

Interactive systems

The next class of systems are systems where a human user interacts with the system in the sense that the user gives a command, the system processes it and presents the result, the user issues another command, and so on. Typical examples are window systems, word processors and other desktop applications. Here, the response time of the system must not be too long if the interaction should work well. If the system takes seconds or more to respond to the users commands, the user tends to be annoyed, but as long as the response times are of the same order as the human response time — typically one or two tenths of a second — the system is perceived to respond instantly, and delays up to half a second are usually tolerable. Therefore, while interactive systems have some degree of real-time requirements, they are quite relaxed and also, the consequences of excessive delays are merely an inconvenience.

Real-time systems

Computer systems that interact with external electrical or mechanical devices or communicate via some shared medium typically have tighter timing requirements. The term *real-time systems* is used to denote systems where timeliness is required for correct operation.

Systems that need to meet deadlines in order to function correctly, but where a failure to do so only causes a temporary decrease in the quality of service and does not cause the whole system to fail are called *soft* real-time systems. One example is audio/video systems, where a missed deadline causes a glitch but the playback still continues. Another example is embedded systems, e.g, a computer controlling the electric windows or the cabin lighting in a car, where occasional small delays will not have any severe consequences.

If missing a deadline may cause the whole system to fail, we have a *hard* real-time system. Continuing the car example, the engine control system is a hard real-time system, as it is critical to the operation of the engine that the fuel injection and ignition are performed at exactly the right time.

It is common that embedded systems consist of both hard and soft real-time tasks, and then techniques like e.g. priority based scheduling are used to guarantee that the hard tasks always get the resources they need, possibly at the expense of the soft tasks.

2.1.2 Predictability

A key attribute of proper real-time systems is predictability; if we want to make real-time guarantees, we must know how long each task may take to execute in the worst case, the worst case execution time (WCET). This is one big difference between interactive and real-time systems; in an interactive system, it is the *average case* performance that usually is the most interesting, as the worst case typically is quite unlikely to occur and it is possible to achieve much better performance on a given platform by disregarding the worst case and optimizing for the common case.

In real-time systems, on the other hand, predictability is paramount as the system must not fail even in the unlikely event that the worst case does occur. Therefore, in hard real-time systems it is often necessary to trade off performance for predictability; in the average case we may have a low CPU utilization in order to guarantee that there will be enough CPU time for every process in the worst case.

In order to meet these requirements on predictability, it is necessary to perform worst case analysis on execution time and memory usage and, based on this, do *a priori schedulability analysis* — a theoretical analysis aimed at determining whether it can be guaranteed that a given set of processes always can be scheduled in a way that they meet their deadlines under a given scheduling model. This is a well understood area and the theoretical foundation is well built.

2.2 Scheduling

The scheduling problem is, simply put, this: Given a set of processes that should execute on a shared processor, find an execution order that ensures that all processes meet their deadlines. This can be done in a number of ways. The oldest, which is still widely used in safety-critical systems, is *static cyclic scheduling*; the CPU time is divided into time

slots and then each process invocation is statically assigned to a particular time slot. The run-time scheduling is simple; the processes of each time slot are executed in due order and when the end of the schedule is reached, execution is restarted from the top. As both execution and communication is statically scheduled, it is easy to verify that a schedule will work. The drawback is that it may be difficult to create the schedule and small changes to the processes may require that a whole new schedule is created from scratch. Also, a static schedule may result in low CPU utilization since the execution times of the different tasks are not equal and therefore, there will often be unused time in some of the time slots. If the execution times of the tasks are not constant, the length of the time slots has to be long enough to accommodate the worst case execution time, as tasks may not overrun their time slot. This further decreases the maximum safe CPU utilization.

An alternative scheduling strategy, which adds more flexibility and transfers the low-level scheduling decisions from the programmer to the run-time system is *dynamic scheduling*; the process scheduler dynamically selects which process that should be allowed to execute at any given instant based on whether that process has work to perform and the relative importance compared to other processes in the system. The rest of this thesis will assume dynamic scheduling and now a brief introduction to various approaches to selecting which process that should be run will be given.

2.2.1 Fixed priority scheduling

In a fixed-priority scheduler, a priority value is assigned to each process. If more than one process is ready to execute, the scheduler always gives precedence to the process with the highest priority. Usually, the scheduler also allows *preemption*, i.e., if a process is executing when another process with higher priority becomes ready, the lower priority process will be interrupted in order to allow the higher priority process to execute without delay.

With fixed priority scheduling, it is usually not possible to have 100% processor utilization without missing deadlines. However, due to the strict priorities, such *overload* is handled in a way that lets the high priority processes continue executing unaffected while those with low priorities are delayed. In cases of severe overload, the low priority processes may not get any CPU time at all. This is called *starvation*.

A problem with fixed priority scheduling is how to assign priorities to processes. The most common approach is *Rate Monotonic Scheduling (RMS)*, which says that the shorter the period time a process has, the

higher its priority should be. If priorities are assigned in this way, standard methods for schedulability analysis exist [34, 46]. It can be proved that for an arbitrary number of independent, periodical processes, a RMS system is guaranteed to be schedulable if the total CPU utilization is less than 69%.

2.2.2 Earliest deadline first scheduling

Another approach to dynamic scheduling is *earliest deadline first (EDF)*. Here, instead of assigning fixed priorities to processes, the scheduling is done based directly on the deadlines of processes; the process with the shortest time left to its deadline is scheduled to run. Thus, this strategy requires no scheduling decisions, other than the deadline assignment, to be made by the developer — all scheduling decisions are taken by the scheduler, at run-time.

An interesting property of EDF scheduling is that 100% CPU utilization is possible and, thus, EDF scheduling is optimal in the sense that if the system is not schedulable using EDF, it will not be schedulable using any other scheduling strategy. However, the handling of overload is drastically different from a fixed priority scheduler; in an EDF system, if the requested CPU utilization is greater than 100%, all processes will miss their deadlines. In effect, the period times will be scaled so that the CPU utilization is 100% and this may be fatal to processes with hard deadlines.

2.2.3 Co-existence of hard and soft processes

In order to overcome the problems with handling overload, especially in EDF scheduled systems, techniques for letting hard real-time processes run with guaranteed deadlines while process with soft or no deadlines may be delayed in order to keep the total CPU utilization at a safe level have been developed.

Constant bandwidth servers

One approach to handling the problem with running both deterministic and non-deterministic processes on the same processor using EDF scheduling is is the constant bandwidth server (CBS) model [1]. For each process or group of processes, a limit on the maximum fraction of the CPU time, the CPU *bandwidth*, is assigned and this is enforced by the scheduler: If a server has used up its CPU quota in the current period it is delayed. A set of constant bandwidth servers running on a single

CPU can be viewed as a if each process were running on dedicated CPU with a given fraction of the original CPU speed.

The CBS model combines the advantages of fixed priority and EDF scheduling; it is possible to guarantee that the hard real-time processes always meets their deadlines by isolating them from non-deterministic processes while still allowing 100% CPU utilization.

Feedback scheduling

Another approach to handling non-determinism is based on that the main goal is to optimize the resulting quality of service rather than some aspect of scheduling like, for instance, minimizing the number of missed deadlines. By using feedback control theory, the scheduling parameters are automatically adjusted at run-time in order to keep the CPU utilization at a safe level while optimizing the quality of service of the application. This is called *feedback scheduling* [2, 12, 13]. One area where this approach is useful is control systems, where it has been shown that the total quality of control can be dramatically increased if the real-time requirements are relaxed.

Figure 2.1 shows the structure of a basic feedback scheduler. A set of tasks generate jobs that are passed to a run-time dispatcher. The execution times of the jobs and the total CPU utilization, U, are measured. Based on this, the scheduler adjusts the period times of the tasks, T_i, in order to keep the CPU utilization at the setpoint, U_{sp}.

If a system contains both hard and soft real-time tasks, it is reasonable that the CPU utilization of the soft processes should be decreased more than that of the hard processes. This can be done by using *elastic scheduling* [11], where a stiffness value is assigned to each process and the scaling of period times is done in proportion to that value.

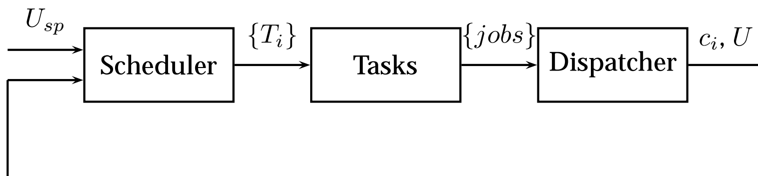

Figure 2.1: *The structure of a basic feedback scheduler.*

2.3 Memory management

The oldest form of memory management is *static memory management*, where the space required for all variables and data structures of the program is allocated statically by the programmer or compiler. As with all static techniques, this makes it easy to verify that a program will work and requires no run-time decisions regarding memory management but the limitations are severe when it comes to writing programs that e.g., build dynamic data structures depending on input.

Dynamic memory management [30, 51] overcomes these limitations by making it possible to allocate memory at any time in the program. However, this comes at the cost of having to manage memory at run-time; when the program wants to allocate more memory, the run-time system must find a suitable space in memory where the requested object will fit. As the amount of physical memory is limited, it is also necessary to reuse the memory occupied by objects that will no longer be used. This can be done manually, by explicitly inserting instructions to deallocate a certain memory area (as `free` and `delete` in C/C++) in the code or automatically by the run-time system.

There are two major problems with manual memory management; failing to deallocate objects that will no longer be used, causing *memory leaks* and deallocating objects too soon, causing *dangling pointers*. The effects of the former is obvious: Failure to deallocate objects that are no longer needed causes excessive memory usage and may cause the system to run out of memory. The latter problem, dangling pointers, is more insidious. It arises when one part of the program deallocates an object, O_1, that is still used by another part of the program. The memory occupied by O_1 may then be used to allocate a new object, O_2. Then, the situation where one part of the program modifies O_1 and another part modifies O_2 may arise. As both O_1 and O_2 refer to the same address, this will result in memory corruption and failure.

The problem of determining when an object should be allocated in order to avoid both memory leaks and dangling pointers is non-trivial in a complex system, and rigid coding conventions and protocols for how pointers may be passed are required. Another way to handle these problems is to let the the run-time system keep track of when an object is no longer reachable and can be deallocated and thereby freeing the programmer from this complex and error-prone task. This is called *automatic memory management* and the techniques used to reclaim unreachable objects are called *garbage collection (GC)*. Examples of early programming languages that use garbage collection are LISP [35] and Simula [15].

2.3.1 Garbage collection

There are different approaches to implementing GC [27]. In this thesis, we will focus on *tracing* collectors — collectors that traverse the reference graph in order to determine which objects are live and which are not. Examples are mark-sweep [35] and copying [36, 20] collectors. Another approach to garbage collection is *reference counting* [14], where the idea is to keep a count of how many references there are to each object and reclaiming objects when the reference count reaches zero.

Most (tracing) garbage collectors need to make multiple passes in order to identify the live objects and reclaim the garbage. For example, a mark-sweep collector first scans all root pointers[1], then traverses the pointer graph starting at these roots, marking all object it encounters and finally *sweeps* the heap, reclaiming the memory occupied by unmarked objects. This can be followed by a compaction phase, where the live objects are moved to one end of the heap in order to form a contiguous area of free memory. We call all the activities required to identify and reclaim garbage a *GC cycle*. E.g., in the mark-sweep-compact case, a GC cycle consists of root scanning, pointer traversal, sweeping and compaction.

It should be noted that during some of the phases (e.g., root scanning and pointer traversal), performing GC work does not cause any memory to be reclaimed. Thus, a generic GC model must assume that no memory is reclaimed until at the end of the GC cycle. Compacting or copying garbage collectors typically have this behaviour, whereas a non-compacting mark-sweep frees memory continuously during the sweep phase.

In the first systems with automatic memory management, the application program (*mutator*) allocated memory until there was no more free memory. Then, the mutator was suspended and the garbage collector performed a full GC cycle, reclaiming the unused memory. This is commonly known as *stop the world* garbage collection, as the whole application is stopped when the garbage collector is running. Another term is *batch GC*. The obvious drawback of batch GC, from a real-time perspective, is that the GC pauses, although infrequent, may be very long, which is unacceptable in a system with hard timing constraints.

[1]The *roots* of the object graph are objects that are, by definition, live. The roots are identified through *root pointers* — pointers located outside the garbage collected heap that reference objects on the heap. Typical examples are pointers located in global variables or variables on the stack.

2.3.2 Incremental and real-time GC

Research within the field of incremental and real-time garbage collection has been going on since the late sixties [9, 49, 50, 16, 7]. The earliest attempts to implement non intrusive garbage collectors used a technique called incremental GC. Here, the GC work is split into a number of very small increments which can be performed interleaved with the execution of the application. In order to guarantee progress of the garbage collector, a number of increments of GC work are performed in connection with each memory allocation request. An example of such an algorithm is Baker's algorithm [7]. Let F_{min} denote the minimum amount of memory available for allocation during a GC cycle, a denote the amount of memory requested, and W_{max} denote the maximum amount of GC work (according to a given metric and corresponding unit) that might be required to complete a GC cycle. Then, the size w of the GC work increment that must be performed in connection with the allocation in order to guarantee that we do not run out of memory before the GC cycle is complete is:

$$w \geq W_{max} \cdot \frac{a}{F_{min}} \quad (2.1)$$

Incremental GC triggered by allocation requests has at least two major disadvantages. Firstly, even if the overhead incurred by a single GC increment is small, a burst of allocation requests can lead to long accumulated delays. Secondly, in order to keep the cost of each GC increment within a low upper bound we might need to use a complex GC work metric in order to decide when to end each increment, since a simple metric often gives a poor approximation of the temporal behaviour of the garbage collector. For instance, if a metric based on measuring the number of evacuated objects in a copying garbage collector is used , an increment which should be short according to the metric can take a long time to perform. The problem is that we might have to scan a significant amount of pointers in order to find just one object to evacuate. Thus increasing the performed amount of work according to the metric by one unit may require a virtually unbounded amount of time.

Performing GC at the time of allocation does make it easy to prove that the garbage collector will always keep up with the application, but it also means that it suffers from the inherent problem of GC work always being performed when application threads run — thus causing interference. The problem of GC work always being performed when application threads run can be overcome by making the GC work *concurrent*, i.e. assigning the GC work to a separate GC thread executing

in parallel with the application threads. This is a strategy applied by a number of garbage collectors, e.g. the Appel-Ellis-Li collector [5], but it has not been much used in real-time settings. Typically, no provision is made for guaranteeing that the collector keeps up with the allocation demands of the application.

In order to satisfy the demands of hard real-time systems, a technique must be found to schedule the GC work of a concurrent GC such that the application is guaranteed to meet all of its hard deadlines. Such a scheduling technique was presented by Henriksson in [22]. That work focuses on embedded systems which are assumed to have a number of high-priority (typically periodic) threads that must meet hard deadlines. It can be observed that in most embedded systems, a relatively small number of such threads exist. Apart from these, low-priority (periodic or background) threads are often executing with more relaxed deadline requirements. This leads to the fundamental idea of Henriksson's work, which is as follows: Do not perform any GC work when the high-priority threads are executing. Instead, assign the work motivated by high-priority allocations to a separate GC thread which is run when no high-priority thread is executing. When invoked, it performs an amount of GC work proportional to the amount of memory allocated by the high-priority threads. Since the garbage collector may temporarily get behind with its work in this way, there must always be an amount of memory reserved for the high-priority threads. Slightly modified generalized rate monotonic analysis [46] can be used both for calculating the amount of memory which need to be reserved and to verify that the garbage collector thread will always keep up with the high-priority threads. Garbage collection work motivated by low-priority threads are performed incrementally at allocation time. Since GC work is partly performed concurrently and partly incrementally in such a system the approach is called *semi-concurrent scheduling*. A system using this scheduling strategy can be described as having three levels of priority:

1. High priority processes
2. Garbage collection required to satisfy the high priority processes
3. Low priority processes and incremental garbage collection

Figure 2.2 shows how the CPU time will be used in a system with one periodic high priority process and one low priority process.

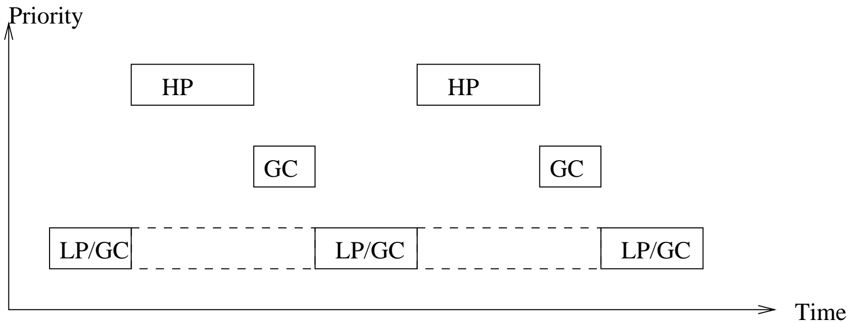

Figure 2.2: *Dividing the CPU time between processes. The system consists of one periodic high priority process (HP) and one low priority process (LP). Whenever a high priority process is suspended, and no other HP process is eligible for execution, the garbage collector (GC) is run. GC work is also interleaved with the low priority process using traditional incremental garbage collection.*

The effect of this scheme is that it makes it possible to guarantee hard real-time performance for threads that actually require it in a system scheduled by a fixed-priority scheduler. Since garbage collection work is not performed while high-priority threads run we can allow ourselves to use a more course garbage collection work metric without affecting real-time performance. An unnecessarily conservative metric will only prevent low-priority threads without hard deadlines to execute as often as they would prefer.

The approach still has some drawbacks, however. One drawback is that it is not immediately suitable for systems with EDF schedulers. Another drawback is that we always have to do an amount of scheduling analysis in order to tune the collector to a specific target platform.

CHAPTER 3

TIME-TRIGGERED GARBAGE COLLECTION

Traditionally, incremental garbage collectors have been scheduled based on the allocations of the application — for each unit of allocation, a corresponding amount of garbage collection work is performed. In this work, a different approach where we use time, instead of allocation, as the trigger for GC work is proposed. That is, garbage collection is scheduled to make the GC cycle finish at a certain time, rather than after a certain amount of allocation.

In [44] the idea of time-based GC and having a fix GC cycle length was introduced. That made it possible to determine how much memory will be allocated during a cycle or to reserve a certain amount of memory for the next cycle while still making it possible to perform schedulability analysis and give real-time guarantees on the run-time system in a straight-forward manner. In that work, we used a hybrid approach, with time-triggered GC that was scheduled using a traditional work metric in a fixed-priority scheduled system.

This chapter presents time-triggered garbage collection more thoroughly. It is argued that time should be used as the unit for garbage collection work and that this is practically feasible. It is shown how the GC cycle time can be calculated in order to guarantee enough GC progress. It is discussed how the process scheduling strategy affects a time-triggered GC scheduler and shown how time-triggered GC can be used to achieve the same objectives using a deadline-based scheduler as the semi-concurrent scheduling strategy does in a fixed-priority system.

The main areas where time-triggered garbage collection scheduling has impact are:

Concurrent GC in deadline-based systems: In order to schedule GC in a way that we can give real-time guarantees while still disturbing the mutator (application) threads as little as possible in a deadline-based system, we want to be able to schedule the GC just as any other thread. With time-triggered GC, this property is inherent in the model, as the only scheduling parameter is the deadline, and we explicitly specify the deadline of each garbage collection cycle.

GC work metric concerns: A traditionally scheduled incremental GC relies on some kind of work metric to determine whether it is in sync with the mutator or needs to perform more GC work. Therefore, such a GC relies on the accuracy of the metric and using a poor metric may cause poor real-time performance. Errors caused by a poor metric can be avoided by using the optimal GC work metric — the actual CPU time required to complete a GC cycle. Additionally, with time-triggered GC, the actual scheduling is independent of the work metric[1] and thus a poor metric does not affect the real-time properties of the run-time system. This allows us to separate the problems of schedulability analysis[2] and run-time scheduling.

Bursty allocation: Applications often show bursty allocation patterns. This means that an allocation-triggered GC would have a bursty execution pattern. Time-triggered GC scheduling does not have this problem as GC work is scheduled so that each GC cycle finishes before its deadline, regardless of when the application performs its allocations.

Unified GC scheduling: Garbage collection schedulers based on a traditional GC work metric are tightly coupled to the actual garbage collector implementation. By using a time-based approach to GC scheduling, it would be possible to separate the GC scheduler from the GC algorithm; using time as both the trigger and the GC work metric provides a simple interface between the GC and the scheduler. Also, as time is easy to measure directly, time-based GC scheduling fits very well into a feedback scheduling framework.

[1] This is not the case for semi-concurrent scheduling, see Section 3.3.

[2] That, of course, still requires worst-case execution time analysis.

3.1 GC cycle time calculation

With time-triggered garbage collection, there is no direct connection between the GC scheduling and the application, so the GC cycle time is the only parameter that controls the progress of the garbage collector. Thus, a time-triggered GC needs correct (or conservative) cycle time estimates in order to make real-time guarantees as each garbage collection cycle must be completed before the application runs out of memory. This section shows how an upper bound on the GC cycle time, which guarantees that the application never runs out of memory, can be calculated.

The following symbols will be used in this section: period time (T), frequency (f), heapsize (H), total amount of allocated memory on the heap (A), amount of memory allocated during this cycle (a), free memory (F), live objects (L), floating garbage[3] (G), amount of memory reclaimed this cycle (r), the set of threads ($\mathbb{P}$), and the allocation per period of thread j (a_j).

Lemma 1 ***For a set of processes, $\mathbb{P}$, with frequencies f_j, allocation requirements of a_j bytes per period and F bytes of memory available at the start of the GC cycle, an upper bound on the GC cycle time that guarantees that the cycle will be completed before the available memory is exhausted is***

$$T_{GC} \leq \frac{F - \sum_{j \in \mathbb{P}} a_j}{\sum_{j \in \mathbb{P}} f_j \cdot a_j} \tag{3.1}$$

Proof A GC cycle must finish before the available memory at the start of the cycle has been allocated. That is,

$$a = \sum_{j \in \mathbb{P}} \left\lceil \frac{T_{GC}}{T_j} \right\rceil \cdot a_j \leq F \tag{3.2}$$

where the ceiling is to cover the worst case schedule. A stronger condition is

$$\sum_{j \in \mathbb{P}} \left(\frac{T_{GC}}{T_j} + 1 \right) \cdot a_j \leq F \tag{3.3}$$

Substituting $f_j = \frac{1}{T_j}$ we get

$$\sum_{j \in \mathbb{P}} (T_{GC} \cdot f_j + 1) \cdot a_j \quad =$$

[3] Floating garbage is objects that are no longer reachable by the mutator but are still believed to be live by the collector. For example, objects that die shortly after they have been marked will not be reclaimed until in the next GC cycle.

$$\begin{aligned}\sum_{j\in\mathbb{P}} T_{GC}\cdot f_j\cdot a_j+\sum_{j\in\mathbb{P}} a_j &= \\ T_{GC}\sum_{j\in\mathbb{P}} f_j\cdot a_j+\sum_{j\in\mathbb{P}} a_j &\leq F \end{aligned} \tag{3.4}$$

$$\therefore \quad T_{GC}\leq \frac{F-\sum_{j\in\mathbb{P}} a_j}{\sum_{j\in\mathbb{P}} f_j\cdot a_j}$$

□

The amount of free memory needs some further discussion. Since any incremental garbage collector suffers from the problem of floating garbage, we must take that into account when calculating the worst case amount of memory available at the start of a GC cycle (F_{min}). Or put differently, we may not be able to use all the free memory during a cycle if we want to be sure that there is also enough memory for the next cycle as the amount of memory that is reclaimed by the garbage collector can vary from one cycle to another due to floating garbage. Let us now examine floating garbage in more detail.

Lemma 2 *Let a^n be the amount of memory that is allocated during the nth GC cycle and L_{max} be the maximum amount of live memory. Then, the sum of live memory and floating garbage at the start of cycle $n+1$ satisfies the inequality*

$$L^{n+1}+G^{n+1}\leq L_{max}+a^n \tag{3.5}$$

Proof Let δ^n be the net change in live memory during cycle n:

$$L^{n+1}=L^n+\delta^n \tag{3.6}$$

Let u^n be the amount of memory that becomes unreachable during cycle n. Then,

$$\delta^n=a^n-u^n \quad\Longrightarrow\quad u^n=a^n-\delta^n \tag{3.7}$$

which gives

$$\left.\begin{array}{l} G^{n+1}\leq u^n=a^n-\delta^n \\ L^{n+1}=L_n+\delta^n \end{array}\right\} \quad \Longrightarrow L^{n+1}+G^{n+1}\leq L^n+a^n \tag{3.8}$$

But $\forall n, L^n\leq L_{max}$, which concludes the proof. □

In order to make hard guarantees, we must determine the maximum amount of memory that can be allocated during a GC cycle without risking that the system runs out of memory due to floating garbage.

Lemma 3 *Let H be the heapsize and L_{max} be the maximum amount of live memory. Then, the maximum amount of memory that can be safely allocated during a GC cycle is*

$$a_{max} = \frac{H - L_{max}}{2} \tag{3.9}$$

Proof The heap contains allocated and free memory

$$H = A + F = L + G + F \tag{3.10}$$

and therefore,

$$F = H - (L + G) \tag{3.11}$$

Applying Lemma 2 to (3.11) gives that, at the start of any GC cycle,

$$F \geq H - (L_{max} + a_{max}) = F_{min} \tag{3.12}$$

Thus, the worst case occurs when $L = L_{max}$, and the remainder of the proof makes this assumption. Then the system has to be in steady state[4] and the maximum amount of floating garbage during a worst case cycle is

$$G^{WC}_{max} = a_{max} \tag{3.13}$$

An upper bound on the amount of memory allocated during a GC cycle must, of course, not be greater than the minimum amount of available memory so the trivial bound is $a_{max} \leq F_{min}$. We will now prove the equality. Objects that are floating garbage at the start of cycle n will have been reclaimed by the start of cycle $n + 1$, which means that

$$F^{n+1} \geq G^n \tag{3.14}$$

The amount of available memory at the start of cycle $n + 1$ is

$$F^{n+1} = F^n - a^n + r^n \tag{3.15}$$

Cycle n is a worst case cycle ($F^n = F_{min}$) iff the amount of floating garbage at the start of the cycle is at the maximum ($G^n = G^{WC}_{max}$). In the worst case, $r^n = G^n$, which corresponds to equality in (3.14). Applying this to Equation (3.15) gives

$$F^{n+1} = F_{min} - a^n + G^{WC}_{max} = G^{WC}_{max} \quad \Longrightarrow \quad a^n = F_{min} \tag{3.16}$$

[4] I.e., for each allocated object, another object becomes unreachable.

Consequently, we can allocate all available memory during a worst case cycle while still guaranteeing that the amount of available memory at the start the following cycle is no less than F_{min}. I.e.,

$$a_{max} = F_{min} \tag{3.17}$$

Finally, equations (3.12) and (3.17) give

$$a_{max} = \frac{H - L_{max}}{2}$$

□

Because the amount of floating garbage may vary, depending on how the execution of the application and the garbage collector are interleaved, the amount of memory reclaimed will also vary from cycle to cycle. Therefore, we cannot always allocate all of the available memory if we want to guarantee that the system never will run out of memory. Consequently, the length of the garbage collection cycles must be calculated based on the worst case amount of available memory.

Theorem 1 *An upper bound on the GC cycle time that guarantees that we always will have enough memory available for allocation is*

$$T_{GC} \leq \frac{\frac{(H - L_{max})}{2} - \sum_{j \in \mathbb{P}} a_j}{\sum_{j \in \mathbb{P}} f_j \cdot a_j} \tag{3.18}$$

Proof The theorem follows from lemmas 1 and 3. □

For an example of how varying amounts of floating garbage affects the amount of available memory, see Figure 3.1. Note that, somewhat counter-intuitively, the dangerous case is when there is *less* than the worst case amount of floating garbage, as this could lead to a situation where we allocate too much memory if care is not taken to avoid that.

It may seem that the limit on the amount that may be allocated during a garbage collection cycle may cause unnecessarily low memory utilization but this isn't the case; the limit on the amount of memory that may be allocated during a GC cycle expressed in Equation (3.9) only affects the cycle time calculations. It is true that in the best case (when we have no floating garbage) at most half of the available memory is allocated during a cycle, but this has nothing to do with the total memory utilization. If the GC cycle time is reduced, the amount of allocation per cycle — and, consequently, the maximum amount of floating garbage — is also reduced. This means that if both high allocation rates and high memory utilization is required, the GC cycles will be short, but as long as $L_{max} < H$ and there is enough CPU time to accommodate both application and GC, the system is guaranteed to work.

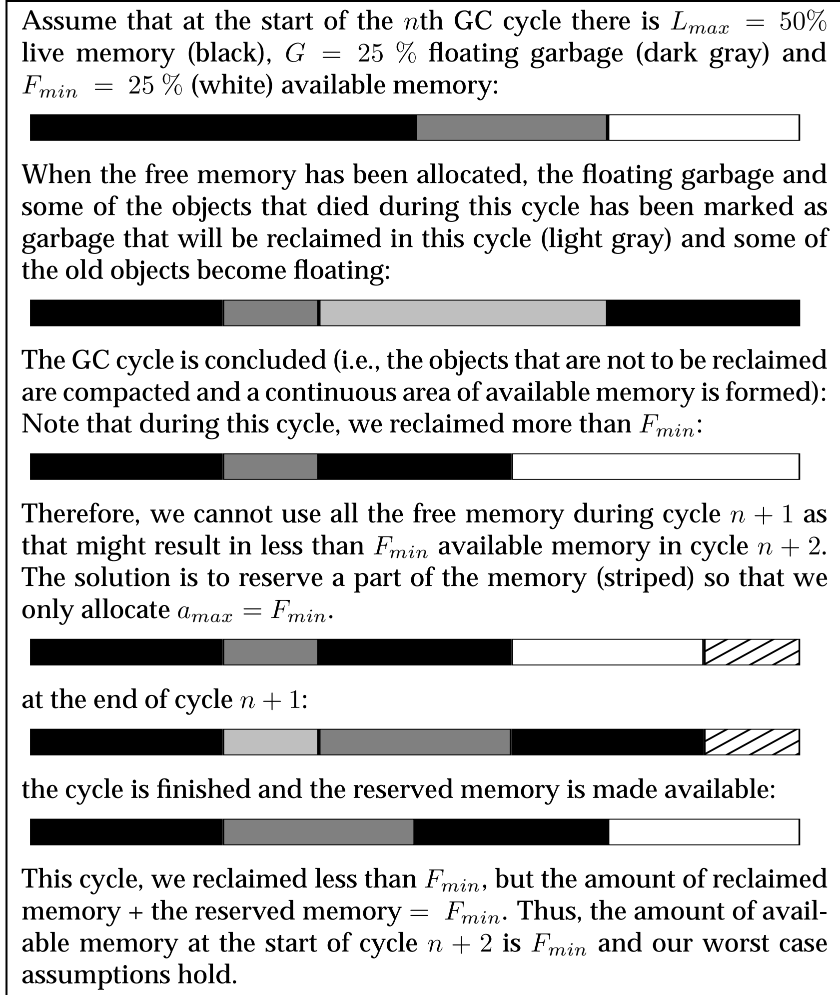

Figure 3.1: *Example of a how the amount of floating garbage may vary between cycles and how our reservation strategy guarantees that there always will be at least F_{min} available memory at the start of a cycle.*

3.2 Using time as the GC work metric

The purpose of a GC work metric is to use quantities that can be directly measured to approximate the temporal behaviour of the garbage collector as closely as possible. However, somewhat surprisingly, the real-time GC literature does not pay much attention to work metrics, and is often content with using some high level abstraction, e.g., the number of "scanned objects", to measure GC progress. Scanning the heap is defined as doing all the GC work to complete a GC cycle. Thus, for a multi-pass GC, like for instance a mark-sweep collector, scanning involves both the mark and sweep phases. This is a way of dodging the metric problem altogether, as it does not define which quantities that should be measured in order to calculate the GC work.

When studying incremental garbage collectors without hard real-time requirements, the focus is on ensuring GC progress while keeping the average GC pause time reasonably short. In a traditional, allocation triggered garbage collector, when garbage collection work is performed in conjunction with each allocation and in proportion to the size of the requested object, it is enough to prove that the metric is conservative. Unfortunately, when applying the same incremental techniques to real-time systems, it is not enough that the GC work metric is conservative; if we want upper bounds on GC pause times, we must also have upper bounds on how conservative the work metric is.

As an example, let us examine a common work metric used in copying collectors: the amount of evacuated memory. Let ΔB denote the amount of evacuated memory (the position of the evacuation pointer relative to the start of tospace) and E_{max} the maximum amount of memory that may need to be evacuated. Then, the amount of performed work, W, and the maximum amount of work during a cycle, W_{max} is

$$\begin{aligned} W &= \Delta B \\ W_{max} &= E_{max} \end{aligned}$$

respectively. With that metric, computing W is trivial and so is ensuring progress if, for each allocation, a corresponding number, according to Equation (2.1), of objects are evacuated.

Unfortunately, this metric does not model the temporal behaviour of the garbage collector very well. For each allocation, an amount of garbage collection work, according to the metric, has to be performed. However, since GC progress is measured in the amount of evacuated objects, any GC activity that doesn't cause new objects to be evacuated will not be captured by the metric. For example, tracing objects that

only contains pointers to already evacuated objects will not increase W. In a worst case scenario, evacuating one single object may require scanning all remaining objects on the heap. Thus, what seems like a small increment under this metric may take a virtually unbounded amount of actual CPU time to perform. This shows how this metric may, in the worst case, cause an incremental collector to have a temporal behaviour close to that of a batch GC and thereby rendering it unsuitable for use in real-time systems.

This problem is described in [22], and Henriksson presents an improved evacuation pointer metric which also takes scanning of objects and roots as well as initialization of reclaimed memory into account. The problem with such a fine-grained metric is that it is much more dependent on details in both garbage collector implementation and application, and therefore requires some amount of manual tuning in order to give good approximations of the CPU time required to perform a certain amount of GC work.

As the purpose of a GC work metric is to approximate the execution time required to complete a GC cycle as closely as possible, the optimal GC work metric is the actual execution time used and this is the approach chosen here; using time as both the trigger for the garbage collector and as the GC work metric (I.e., the total GC work of a cycle is the CPU time the system has to spend on performing garbage collection.) in the actual run-time system. This has, to our knowledge, not previously been done.

By using time as the GC work metric, the amount of performed work can be measured directly, which eliminates all errors in the performed work metric. The total amount of CPU time required to complete a GC cycle, has to be calculated using standard worst case execution time analysis techniques[5]. Then the GC scheduling will be independent of both the application and GC implementation and the problems with bursty allocation patterns and imperfect GC work metrics are avoided. An additional advantage is that no assumptions about the GC algorithm, implementation or application behaviour are hard-wired into the GC work metric[6].

[5] Note that this requirement is no restriction in relation to traditional real-time garbage collection techniques; if we want to be able to make hard real-time guarantees, we have to do worst case analysis. If this is not possible, it may be better to use some adaptive technique, as described in Chapter 4.

[6] Of course, these aspects affect the GC workload and has to be taken into account when calculating the GC workload, but having a generic metric allows us to separate e.g., the GC scheduler from the GC algorithm.

Another important result of using CPU time as the GC work metric is that the GC work calculations are made on a per cycle instead of a per increment basis. Thus, if the W_{max} estimates are conservative, the additional overhead will be distributed evenly across the GC cycle instead of causing individual increments to be too long as described above. Hence, using time as the GC work metric helps mitigate the negative effects of using a conservative GC work metric when using an incremental GC.

Also, using execution time as the GC work metric together with time-triggered garbage collection scheduling makes it easier to integrate the GC scheduling with the application process scheduler, since the two scheduling parameters, execution time and deadline, are explicit in the model. Thus, the GC thread can be scheduled like any other thread in EDF as well as fixed-priority systems. It also fits well into a feedback scheduling system, as it makes the execution time requirements of the garbage collector explicit. Finally, it has the advantage that it makes it possible to incorporate other factors that affect the GC execution time, but are not directly tied to the garbage collection algorithm (e.g., caches, pipelines, etc.) into the GC work calculations and measurements.

3.3 Scheduling

This section discusses how time-triggered GC scheduling can be implemented in fixed priority and deadline based systems, respectively and how the general process scheduling policy affects the garbage collection scheduling. It also relates time-triggered GC scheduling to semi-concurrent scheduling and handling of background tasks.

Based on the cycle time calculations presented in Section 3.1, we can use standard scheduling techniques (e.g., RMS or EDF) and schedule the GC as any other thread since the scheduling of individual GC increments is implicit; the only real requirement is that the GC cycle has ended and enough memory is made available before the application runs out of memory. As the deadline is the sole scheduling parameter, this means that the GC work calculations are only needed for schedulability analysis and not for ensuring GC progress at run-time. Hence an error in the metric alone cannot cause the GC to run too slowly, which gives a more robust system. If the system is schedulable, the GC will finish on time, without causing any other thread to miss its deadline.

In systems where hard real-time tasks co-exist with background tasks without timing requirements, we want hard guarantees that the GC always will make memory available to the real-time tasks on time but we also want to avoid unnecessary disturbance of the background tasks.

Conversely, we want to protect the GC from the background tasks in the sense that allocations performed by a background task must not cause the GC to miss its deadline or fail to make enough memory available. These problems are addressed by the semi-concurrent GC scheduling strategy. The effects of incorporating time-triggered GC and semi-concurrent scheduling will now be examined.

When implementing a semi-concurrent garbage collector under the aforementioned scheduling policies, the main difference is that in a fixed priority system we must explicitly schedule each GC increment in order to spread the garbage collection overhead evenly across the cycle. That is, each time the garbage collector is invoked, it has to determine how long that increment should be (according to the metric used) and, when enough work has been performed, the GC must suspend itself until the next increment is triggered. Otherwise, the garbage collector thread might starve low priority threads for long periods of time. In an EDF system, the scheduling of GC increments can be left to the process scheduler, as there are no fixed priorities and, thus, no risk of starvation.

A consequence of the requirement that the garbage collector must determine the length of each increment is that the actual scheduling will depend on both the cycle time and the work metric. In an EDF system, the only scheduling parameter is the deadline, and the garbage collection thread can be scheduled like any other thread. Therefore the run-time scheduling is independent of the work metric and worst-case analysis, which is a big advantage in practice, as worst-case analysis often is based on measurements rather than exact analysis.

A problem with using allocation-triggered, concurrent GC in hard real-time systems is that it is necessary to reserve a certain amount of memory for allocations of the high priority processes. Without a safety margin it is impossible to guarantee that schedulability will not be jeopardized due to special effects near the end of GC cycles [22].

The reason that a safety margin is required is that when using fixed-priority scheduling, the garbage collector is never allowed to interrupt a high priority thread. Without a safety margin, the system could reach a state when there is memory left (and, thus, the cycle not yet finished) but not enough memory for all of the allocations of a high priority thread during its execution. Since GC work is suspended during the execution of high priority threads, activating a high priority thread at such an instant would cause the system to run out of memory which, in turn, causes “panic” stop-the-world GC. Therefore it was necessary to reserve enough memory for the worst case allocation requirements of the HP threads during the maximum response time of the GC thread.

With time-triggered GC, on the other hand, this would not be a problem. As the deadline of the GC thread is explicit in the model, traditional schedulability analysis could be performed and the safety margin would not be necessary.

3.3.1 Fixed priority scheduling

In a fixed priority system, a higher priority thread always get precedence over lower priority threads. Therefore, a semi-concurrent GC must spread the GC work evenly across the whole cycle and not do more work in each increment than absolutely necessary, in order to avoid subjecting threads that run with a lower priority than the GC thread to unnecessary starvation and excessive jitter. Thus, some GC work metric has to be used to determine if the garbage collector has made enough progress.

Naturally, for a given GC cycle time, T_{GC}, all the garbage collection work required to complete a GC cycle has to be performed before T_{GC} seconds have elapsed. In order to ensure sufficient GC progress, the GC scheduler should maintain the invariant

$$\sum w \geq W_{max} \cdot \frac{t - t_{cycle\ start}}{T_{GC}} \tag{3.19}$$

That is, the fraction of GC work performed should be greater than or equal to the fraction of the cycle time elapsed. This corresponds to Equation (2.1) with time instead of allocations as the trigger, on the right hand side. Scheduling garbage collection according to this invariant ensures that progress will be made at a well-defined rate regardless of if, and when, the application allocates memory.

3.3.2 EDF scheduling

The first property of semi-concurrent scheduling, non-intrusiveness, is inherent in the EDF model; if the requested CPU utilization is less than 100%, all deadlines will be met.

The second property of the semi-concurrent model, isolating the high priority threads from the low priority ones, and thus not having to do worst-case analysis on the LP threads, can in an EDF system be achieved by using Constant Bandwidth Servers (CBS) with the addition of a priority, or importance, attribute for the servers. Then, the HP and LP threads in the semi-concurrent model would correspond to HP and LP servers.

In such a model, the threads running on HP servers would just do allocations without any GC penalty, while the threads on the LP servers

would do incremental GC at allocation time. When incremental GC is performed due to a LP allocation, both the deadline and execution time of the GC thread should be decreased as the memory allocation has reduced the amount of available memory and the incremental GC work has brought the GC cycle closer to its finish. Moving deadlines to an earlier point in time is, however, not allowed in an EDF system in the general case as this causes a temporary increase in the requested CPU utilization and might lead to missed deadlines. This could be solved by temporarily reducing the bandwidth of the LP server with a corresponding amount or, if the remaining CPU time in the LP server's budget is too low, delaying the allocation that would cause incremental GC work until the next CBS period. In practice, however, this is not a problem as the GC cycles typically are much longer than the period times of the application threads and therefore the deadlines and/or server bandwidths can be adjusted at the thread release times when it is safe to do so.

Another way to make sure that the memory management overhead never may cause the critical parts of the application to miss their deadlines is presented in Chapter 5. By introducing priorities for memory allocations, the run-time system is able to automatically prioritize memory allocation requests (i.e., deny non-critical allocations) in order to guarantee that the system will not run out of memory or become unschedulable because of a too high GC workload. In essence, this can be viewed as dividing the application into critical aspects, which are guaranteed to be executed on time and non-critical aspects, which are only executed if it is safe to do so.

3.4 Summary

A new way of scheduling garbage collection work in real-time systems was presented; instead of using allocation as the trigger for GC work, time is used, and instead of ensuring that every GC cycle finishes before all available memory has been allocated, garbage collection is scheduled in a way that gives a fixed GC cycle time.

This approach leads to a number of desirable properties: It makes it easy to spread the garbage collection work evenly across the GC cycle. Consequently, a time-triggered GC does not suffer from the bursty execution pattern, due to the application performing allocations in bursts, that an allocation-triggered GC does.

As the most important scheduling parameter, the deadline, is explicit in the model, a time-triggered GC can be scheduled as any other process in both fixed-priority and EDF systems with real-time requirements.

It is shown how a GC cycle time that guarantees that the application never runs out of memory can be calculated based on the amount of live memory and allocation rate of the application.

The metrics used to measure garbage collection work in previous real-time garbage collectors often fail to model the temporal behaviour of the garbage collector which may cause poor real-time performance. By using time as the GC work metric, such inaccuracies can be avoided, as time can be measured directly. This also makes it suitable for use in a feedback scheduling environment.

CHAPTER 4

ADAPTIVE GARBAGE COLLECTION SCHEDULING

Feedback control is a good way to cope with model uncertainty, and has successfully been used in process schedulers for real-time control systems with non-deterministic execution times. Feedback scheduling is very suitable for systems which changes between different operating modes with different resource utilization patterns where using worst case assumptions would yield an unacceptably low CPU utilization. A feedback/feed-forward system can adapt to the changing requirements of the application and tune e.g. the period times of the threads in order to keep the CPU utilization at a safe level while optimizing the quality of service delivered by the system.

This chapter investigates if and how a time-triggered GC can be made auto-tuning and how it can be incorporated in a feedback scheduling system in order to make the memory management overhead explicit and let the process scheduler take this into account when scheduling the application threads. Section 4.1 gives an introduction to the problem and motivates the work. Section 4.2 gives an overview of the proposed architecture. In order to schedule a task, we need two parameters; its execution time and its deadline. Section 4.3 shows how the cycle time can be estimated and Section 4.4 discusses how the amount of CPU time required to complete a GC cycle can be determined.

4.1 Introduction

Worst case analysis is, in the general case, difficult even for relatively small programs and for a garbage collector it can be even harder, as the execution time of the garbage collector not only depends on the applica-

tion but also on the thread scheduling (which affects both how the application and GC are interleaved and in what order memory allocations are performed and consequently where on the heap the objects are placed.) Furthermore, the execution time of the memory manager relies heavily on memory performance which is a big source of non-determinism on a modern computer system with caches, etc.

Even if worst-case analysis could be performed it may be quite pessimistic which leads to unacceptably low CPU utilization. Using feedback control, on the other hand, lets us exploit varying resource utilization among the application threads, allowing us to achieve better overall utilization of both CPU and memory.

Manual tuning of GC scheduling parameters is based on certain assumptions about the heap usage pattern of a particular application. Tuning a real-time GC thus requires a great engineering effort and is therefore usually only practically feasible for safety-critical, hard real-time systems with a small number of simple processes and not for larger systems or systems with less rigorous safety requirements.

In order to achieve greater flexibility and allow a larger number of diverse applications to run with adequate performance without requiring huge engineering efforts to tune the GC, we investigate whether it is possible to make the GC scheduler auto-tuning, which would let us run applications with real-time performance without any *a priori* analysis.

We should also not forget that hard real-time guarantees are only as good as the worst case assumptions they are based on so if the worst case estimates are wrong the system will fail even if the scheduling algorithms and GC work metrics used are correct.

Previous work on feedback scheduling and automatic identification of (soft) real-time systems [2] has showed how self-tuning regulators can be used to control resource allocation without a priori knowledge about the task requirements. However, in the existing feedback scheduling systems we are aware of the memory management overhead is either ignored or treated implicitly as a part of the application's execution. It would be desirable to make the memory management overhead explicit in the model in order to make it possible to handle it more efficiently.

The problem with GC scheduling is that the GC has to finish each cycle before the available memory is exhausted or else it will stop-the-world to complete the cycle, which is a bad thing for the hard real-time tasks. Thus, care has to be taken to make sure that the GC is always given the CPU time (or bandwidth) it needs. This implies that we cannot use standard feedback scheduling on the garbage collection thread, as making the GC cycles longer (to reduce the GC's CPU utilization) may be fatal. In the proposed approach, the deadline and CPU utilization calcu-

lated by the GC scheduler cannot be changed by the feedback scheduler, but it must take them into account when calculating the period times of the application threads. This corresponds to a rigid task in [11].

Another approach to keeping the system schedulable is to limit the memory allocation rate. This supplements the auto-tuning of the GC by making it possible to control the allocation rates of the application threads. This is addressed in Chapter 5.

4.2 Overview of an adaptive GC scheduler

The proposed adaptive garbage collection scheduling model consists of two orthogonal auto-tuners; the GC cycle time (deadline) and the GC work (execution time) estimations. The cycle time estimation is used directly to determine the deadline of the GC thread (which is used by the scheduler for the actual scheduling). The execution time estimation is only needed if the GC is to be used in a semi-concurrent system, where it is needed to determine the length of the increments, or in a feedback scheduling system, where the execution time is used to perform the on-line schedulability analysis required to guarantee that the system remains schedulable.

Figure 4.1 shows a block diagram of the cycle time and execution time estimation. In the cycle time estimation, we use a black-box view on the application; the estimates do not depend on any information about the application other than the allocation rate, which can be measured directly[1]. The state of the memory manager, on the other hand, is quite important for the execution time estimation and might therefore be necessary to take into account, either through manual or automatic tuning. Section 4.4 discusses both a black box and a clear box approach to garbage collection work estimation.

Figure 4.2 shows how the garbage collection scheduler fits into a general feedback scheduling system. The GC thread is scheduled as a normal application thread, but with the important difference that it is allowed to set its own deadline whereas the feedback scheduler changes the application threads' deadlines in order to optimize CPU utilization. As mentioned, the special treatment of the GC thread is necessary since the GC will stop all application threads if the system runs out of memory and that must be avoided as it leads to long GC pauses and unacceptable real-time performance.

[1] A clear-box approach is discussed briefly in Section 7.1.

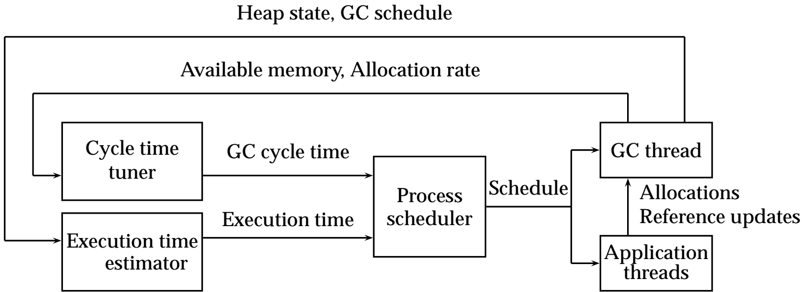

Figure 4.1: *Block diagram of an adaptive GC. Based on measurements of the amount of available memory, the allocation rate of the application, the heap state and the previous execution of GC, the cycle time and execution time of the GC is estimated.*

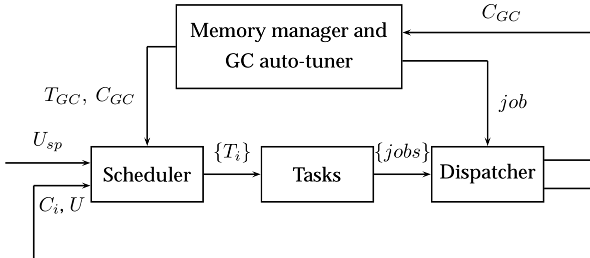

Figure 4.2: *Feedback scheduling of both application tasks and GC. The GC task issues jobs which are dispatched just as any other jobs. The only difference between the GC task and the application tasks is that the GC is allowed to set its own period time while the feedback scheduler changes the application tasks' period times in order to keep* $U \leq U_{sp}$.

4.3 Automatic GC cycle time estimation

As we have seen, the GC cycle length can be calculated at design-time based on the allocation requirements of the high priority threads. If this is not practical for some reason (for instance that the application's execution pattern varies greatly depending on operating mode or that it should be run on many different platforms and we do not want to do analysis for all possible target platforms or even know which platform it will run on) or if we want the GC scheduler to be completely transparent to the developer we have to use some adaptive technique to measure and control the GC scheduling parameters on-line.

A very simple model is to measure or estimate the allocation rate ($\dot{a}$) of the application. We can then calculate at which time all the currently remaining free memory (F) will have been allocated — the GC cycle's deadline.

$$T_{remaining\ this\ cycle} = \frac{F}{\dot{a}} \tag{4.1}$$

which, $T_{elapsed}$ seconds into the GC cycle, gives the cycle time

$$T_{GC} = \frac{F}{\dot{a}} + T_{elapsed} \tag{4.2}$$

As memory allocations typically are bursty, the measurement of the allocation rate may need to be low-pass filtered in some way to keep the deadline estimates more stable, which in turn helps to reduce the update frequency for the scheduling parameters. However, we must never underestimate the allocation rate, as this might lead to an out-of-memory situation. Therefore, we must react quickly to actual changes in allocation rate while avoiding chatter due to bursty allocations.

The simple model of Equation (4.2) performs well if roughly the same amount of memory is reclaimed in each GC cycle but it suffers from the same problems with floating garbage as described in Section 3.1 although the symptoms are a bit different. In the fixed deadline case, the system might run out of memory if the GC cycle time was too long. In an adaptive system, the cycle time will be tuned to ensure that this does not happen so the problem in this case is that the system might become unschedulable.

One example of this that we encountered in our experiments with this simple model is that if there, for some reason, is much floating garbage during one cycle, little memory will be reclaimed during that cycle. Then, the following cycle will have to be very short and we get

a memory trace like the one shown in Figure 4.3. This could cause real-time problems since the required CPU utilization of the GC will be much higher during the short cycles than during the long ones, as the amount of GC work is roughly the same[2] in all cycles, but it has to be done in a much shorter time in the short cycles.

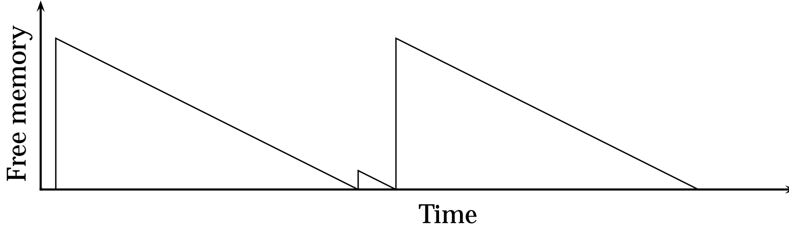

Figure 4.3: *Example of a very short GC cycle caused by large amounts of floating garbage.*

In order to handle the worst case amount of floating garbage, we need to reserve memory so that the allocations during the next cycle can be satisfied even if no objects are reclaimed during the current cycle. Let $\hat{\dot{a}}$ be the estimated allocation rate and $\dot{a}^{next}$ be the allocation rate for the next cycle. Then $T_{GC} \cdot \dot{a}^{next}$ will be allocated during the next GC cycle and we get

$$\hat{T}_{GC} = \frac{F - \hat{T}_{GC} \cdot \dot{a}^{next}}{\hat{\dot{a}}} + T_{elapsed} \tag{4.3}$$

$$\implies \quad \hat{T}_{GC} = \frac{F + \hat{\dot{a}} \cdot T_{elapsed}}{\hat{\dot{a}} + \dot{a}^{next}} \tag{4.4}$$

If we assume that the mutator will continue at the measured allocation rate, i.e., $\dot{a}^{next} = \hat{\dot{a}}$, we get

$$\hat{T}_{GC} = \frac{1}{2}\left(\frac{F}{\hat{\dot{a}}} + T_{elapsed}\right) \tag{4.5}$$

which is the adaptive equivalent to Equation (3.18). If the allocation rate is constant, this means that we should reserve half of the available

[2]Of course, this depends on the garbage collection algorithm as well as on implementation details. However, the execution time of a garbage collector typically depends on both the amount of retained and reclaimed memory. Even algorithms where there is no explicit *free* operation, like for instance a copying collector, have a fraction of the cost that is proportional to the amount of reclaimed memory if, e.g., the initialization of memory is taken into account.

memory at the start of the current cycle for the allocations during the next GC cycle. Doing so guarantees[3] that we can handle the worst case, when all the objects that die during a cycle becomes floating garbage and will not be reclaimed until at the end of the next GC cycle. Figure 4.4 shows how the memory trace of the floating garbage example would look with the reservation strategy in place; the cycles are shorter and the floating garbage anomaly in the first cycle has much less impact on the GC cycle lengths.

As we shorten the GC cycles, the number of GC cycles increase and consequently the incurred GC overhead increases. However, as we do not use all of the heap, the additional overhead is not as big as it would seem.

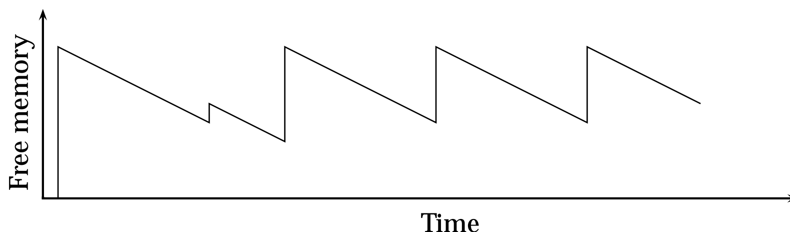

Figure 4.4: *Example of how reserving memory for the next cycle mitigates the problems of floating garbage depicted in Figure 4.3.*

Only allocating at most half of the available memory each GC cycle might seem wasteful, but this is the price we pay for incrementality. Note that this reservation strategy only affects the length of the GC cycles and not the overall memory utilization. If, for instance, the amount of allocated memory is 80% of the heap, the GC cycle length would be set so that 10% of the total memory is reserved for the next cycle.

It should also be noted that a copying collector [7] by design has the property of reserving a part of the available memory for the next cycle so this is only a concern with mark-sweep type collectors and it is not a problem to implement a mark-sweep collector so that it only makes memory available at the GC cycle boundaries. There also exist mark-sweep type algorithms with more than one allocation area that, by design, have this behaviour (e.g., Bengtsson's [8]).

[3] Given, of course, that the total amount of live memory is smaller than the heap-size.

4.4 GC workload estimation

As discussed in Chapter 3, using semi-concurrent GC in a fixed-priority system requires good estimates on the total amount of GC work that must be performed to complete a GC cycle as the scheduling of the increments depend on it. Also, in feedback scheduling systems, online schedulability analysis is performed and some technique is used to limit the maximum allowed CPU time utilization of the application threads if the total requested CPU utilization is too large. Therefore, in such systems, it must be possible to determine how much CPU time that is required in order to complete a GC cycle. This section discusses how the execution time required to complete a GC cycle, given a certain heap state, can be estimated at run time.

It is important that the GC work estimates are not too low since this might cause us to allocate too large a fraction of the CPU time to the application threads, causing the GC thread to miss its deadline, which might, in turn, cause an out-of-memory situation and stop-the-world GC. The estimates should also not be too high in order to avoid unnecessarily low CPU utilization and undue disturbance of low priority threads.

When implementing the estimation, one can choose either a black or clear box approach. A black box model doesn't require any information of the internals of the memory manager and only tries to predict the future execution times based on the history. This has the advantages that it is fairly easy to implement and that it, by design, is independent of the actual garbage collector used.

The drawback is that it cannot take advantage of any information the memory manager has about application behaviour or system state and thus will react poorly to transients. That could be handled by introducing feed-forward of mode changes. That is, the applications could inform the memory manager that they are about to change their memory allocation rate which would allow the GC scheduler to change its GC work estimates to avoid adverse effects at the transients.

The clear box approach requires a more detailed interface between the GC scheduler and the memory management system as well as a dynamic model of the memory system. We want to be able to measure as many parameters (live memory, dead memory, number of live objects, etc) as possible on the heap and find how each of these parameters affect the execution time of the GC using some automatic system identification technique. We also want to take the application behaviour into account by e.g. measuring the allocation rate.

In order to estimate the amount of CPU time required to perform the GC work that needs to be performed in order to finish a GC cycle, there are a number of problems; we need to

Measure or estimate the heap state: In its most simple form, we only measure the amount of available memory. However, to obtain better estimates, the amount of live memory, dead objects and other quantities that affect the execution time of the GC (e.g., the number of pointers that need to be traversed, the number of objects that will be relocated, etc.) must be determined, either through direct measurements, calculations, or feedback control.

Estimate the accumulated GC workload: Based on the heap state, we must estimate how much GC work needs to be performed in order to complete a GC cycle on the heap in its current state. For an empty heap, the accumulated GC workload is zero, and the accumulated workload increases monotonously, due to allocations and pointer updates.

Measure the amount of performed GC work: This can be done in a quite straight-forward manner if we use time as the GC work metric, provided that we have control over the process scheduler and have access to a high resolution timer.

Estimate the total amount of GC work in a cycle: Finally, based on the other estimates, the total amount of work required to complete a GC cycle is estimated.

4.4.1 Estimating accumulated GC workload

This section discusses measuring and estimating the amount of garbage collection work, W, that has to be performed given the heap state at a certain instant. The accumulated GC workload at time t, W_t, is the amount of work the garbage collector would have to do if a new GC cycle was started at time t and the GC was allowed to run exclusively until it had finished that cycle. By definition, W will increase monotonously during the GC cycle (due to object allocations and pointer updates), and we will use the *rate* at which W increases to estimate the total amount of GC work that will have to be performed to complete this cycle.

The GC work can be expressed as a function of the state of the heap

$$W = f(S_h). \tag{4.6}$$

It is not practically feasible to use the state of the heap *per se* when calculating the amount of GC work and therefore an abstract model is required. Objects allocated on the heap are either live or dead, which leads us to the following abstract representation of the heap state:

$$S_h = \begin{bmatrix} \#\ GC\ roots \\ \#\ allocated\ objects \\ \#\ allocated\ bytes \\ \#\ live\ objects \\ \#\ live\ bytes \\ \#\ dead\ objects \\ \#\ dead\ bytes \end{bmatrix} \tag{4.7}$$

The main factor this model fails to capture is the actual placement of the objects on the heap. The placement of objects affect the GC workload since it affects which objects needs to be moved in a compacting collector or the degree of fragmentation in a non-moving GC. However, taking object placement into account would essentially mean using the entire heap itself as the heap state representation.

It is reasonable to approximate the work required to perform a GC cycle with a linear combination of the components of S_h. For instance, the time required to scan root pointers is proportional to the number of roots, the time required to mark all objects are proportional to the amount of live objects, etc. Thus, the GC workload can be expressed as

$$W = A\ S_h \tag{4.8}$$

for some matrix A. We can then identify the coefficients of A using some numerical method like least square approximation.

When we have identified the function f, or the identified coefficient matrix A, the GC work estimate only depends on the heap state, and not on any internal state of the GC. This facilitates the development of a well-defined interface between the memory manager and the GC scheduler, which makes it possible to separate the two problems and, hence, implement a generic GC scheduler that can be tuned to fit different GC algorithms.

This identification requires quite a lot of computation, so it is not feasible to do the full identification at each sample. However, the problem can be divided into two parts: GC algorithm (implementation) dependent and application dependent factors. The former does not change rapidly during execution, whereas the latter may do so, e.g. at mode changes. Thus, a cascade structure can be used: The more demanding

optimization problem of identification of the GC implementation dependent parameters can be solved either ahead-of-time or in a background process while the inner loop that handles the application dependent dynamics can be made much simpler and thus run at a higher rate.

One way to make the inner loop simple is to identify two aspects of the GC work and then express the total amount of work as a linear combination of those two terms.

$$W = \alpha W_x + (1 - \alpha) W_y \tag{4.9}$$

$$W_x = A_x S_h \tag{4.10}$$

$$W_y = A_y S_h \tag{4.11}$$

The inner loop then only has to estimate one parameter, α, which can be done using some simple filtering. The outer loop, that has to find the matrices A_x and A_y requires much more computation, but can be run at a much lower rate. It is also possible to do the identification of the GC implementation dependent parameters ahead of time and not at run-time. Figure 4.5 shows a block diagram of the described auto-tuner. The estimation of the GC cycle execution time ($\widehat{C}_{GC}$) is described in Section 4.4.2.

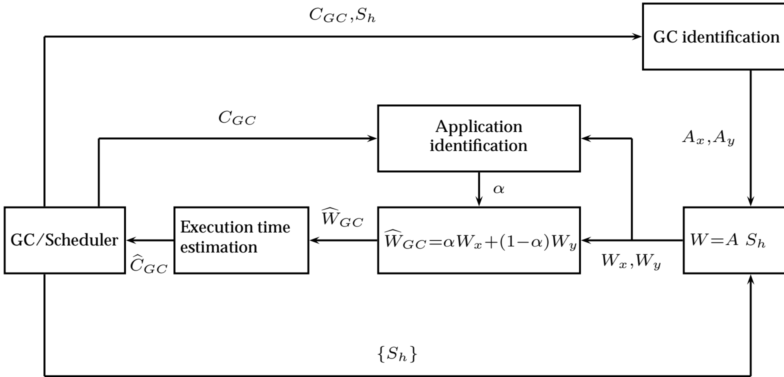

Figure 4.5: *GC work estimation. The application dependent dynamics are captured by the inner loop which estimates the parameter α using simple filtering and is run at a high rate. The GC implementation dependent matrices A_x and A_y are identified using some numerical optimization method which requires more computation but can be run at a lower rate.*

For example, in a mark-sweep collector the GC work could be divided into two phases, mark and sweep. The total amount of GC work required to complete a cycle can then be written

$$W = \alpha \cdot W_{mark} + (1 - \alpha) \cdot W_{sweep} \tag{4.12}$$

with

$$W_{mark} = A_{mark}\, S_h \tag{4.13}$$

and

$$W_{sweep} = A_{sweep}\, S_h \tag{4.14}$$

The division into phases (e.g., *mark* and *sweep*) is common when calculating the amount of GC work using some traditional, ad hoc, metric and fits naturally with the operation of the garbage collector. However, that may not be the best way to capture the variations of the application. Perhaps a division into W_{live} (the part of work that is due to live objects, e.g. tracing and moving) and W_{dead} (e.g., freeing, coalescing, memory initialization, etc.) is better as that more directly reflects the variations in the application which affect the GC workload.

Another division may be into W_{size} (part of work proportional to the amount, in bytes, of the allocated memory) and W_{number} (part of work proportional to the number of allocated objects). The rationale behind that division is that e.g., the time spent moving objects is proportional to the number of bytes that has to be moved, but that the time spent marking the object graph is proportional to the number of live objects rather than their size.

Of course, for any approximative model there is a pathological example which breaks that model, but by using feedback control we can achieve a greater degree of tolerance to model error.

While the proposed division of garbage collection work into two terms, W_x and W_y makes it possible to perform the identification of the application dependent dynamics by estimating a single parameter, α, it is still an open problem how the aspects x and y should be selected. More studies are required and experiments with different partitionings has to be carried out.

4.4.2 Estimating total GC work

Now we need to put it all together in order to estimate the total amount of work required to complete a GC cycle. One way of doing this is to use feedback control with a (time-variant) PD-controller.

By looking at $\frac{dW}{dt}$ (the rate at which the GC work increases) and $\frac{dC}{dt}$ (the rate at which GC work is performed), it is possible to extrapolate

1) how much work that will need to be performed to complete the cycle and *2)* how much that actually will have been performed at the end of the cycle if continuing at the current rate. If it is found that the GC will not be finished in time, the GC work estimate ($\hat{C}$) is increased with the difference (e). If GC is performed at too fast a rate, the work estimate is decreased correspondingly.

This can be expressed more formally as follows: At some point during a GC cycle, at time t; ($t_s < t < t_e$, where t_s is the start time of this GC cycle, and t_e is the finish time.) the amount of work that needs to, and has been performed, respectively, at the end of the cycle is

$$W_e = W_t + t_{remain} \frac{dW}{dt}; \tag{4.15}$$

$$C_e = C_t + t_{remain} \frac{dC}{dt}; \tag{4.16}$$

$$t_{remain} = t_e - t; \tag{4.17}$$

Then, the GC work estimate is

$$\hat{C} = \hat{C} + K \cdot \beta \cdot e \tag{4.18}$$

where

$$e = W_e - C_e \tag{4.19}$$

and

$$\beta = \begin{cases} \beta_{up} & \text{if } e > 0 \\ \beta_{down} & \text{otherwise} \end{cases} \tag{4.20}$$

The scaling by β is used to obtain good adaptive behaviour. Since we want to avoid underestimating the amount of GC work, the estimate is increased faster than it may decrease for a given error. Figure 4.6 shows how the estimation error e in Equation (4.19) is defined.

Figure 4.7 shows a simulation of the $\hat{C}$ estimation. Notice the step at $t = 3200$ and how the feedback handles it; the execution time estimation is increased at a faster rate than it is decreased. Therefore, a step causes the GC to overestimate the work, and thus to perform more GC work than necessary, leading to a premature finish of the GC cycle. This may cause the GC to temporarily starve low priority processes, but ensures that the high priority processes will not be affected (i.e., the GC cycle will finish on time and stop-the-world panic GC will be avoided.) In this example we assume that the GC increments are scheduled according to Equation (3.19). The results would be the same if using earliest deadline first scheduling with CBS and the CPU utilization was 100%.

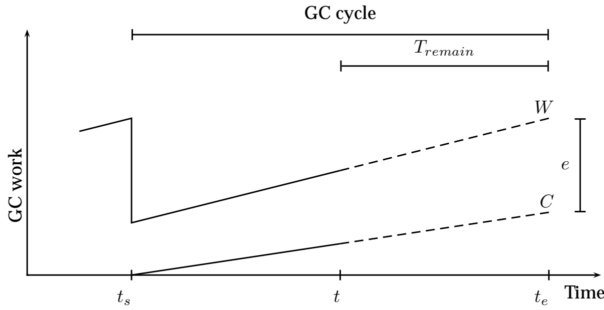

Figure 4.6: *Calculation of GC work estimation error by using $\frac{dW}{dt}$ and $\frac{dC}{dt}$ to extrapolate the values of W and C at the end of the GC cycle. In this example, garbage collection is performed at too slow a rate, and the GC work estimation, $\hat{C}$, has to be increased.*

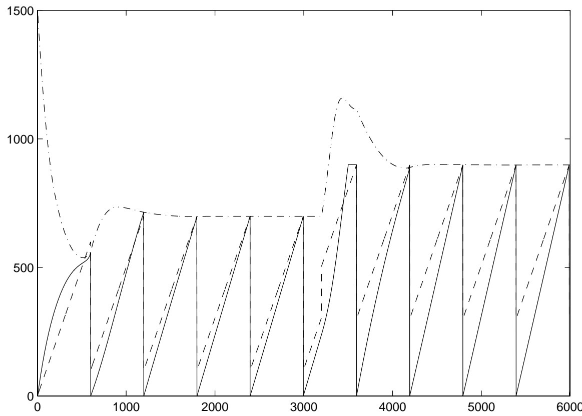

Figure 4.7: *Simulation of the GC work estimation. The estimated GC cycle work, $\hat{C}$ is the dash-dotted line. The dashed line is the accumulated GC workload (W) and the solid line is performed GC work (C). The GC work is scheduled based on $\hat{C}$.*

4.5 Summary

An approach to making a time-triggered garbage collection scheduler auto-tuning based on the observation that we need to estimate the two scheduling parameters deadline and execution time was presented. It was argued that the two problems are orthogonal and techniques for both estimations were proposed and discussed. When using an EDF scheduler, only the deadline estimation is required, and it was found that this can be implemented in a straight-forward way with adequate real-time performance.

The cycle time estimation is based on simply measuring the current allocation rate and, based on this allocation rate, calculating when the currently available memory will have been used up. In order to handle varying amounts of floating garbage, a strategy to reserve a portion of the memory for the allocations of the following GC cycle. Experiments[4] support the feasibility of this approach.

The execution time estimation is more demanding, and particularly to estimate the current GC workload is still an open problem. More studies on how to express the GC work and how to factor the GC work into different phases or aspects have to be carried out.

Given an estimate of the current GC workload, the total work required to complete a GC cycle can be estimated by using feedback control, and simulations support the feasibility of this approach. However, this is preliminary results and the adaptive techniques used in the simulation are quite simplistic, so more work is required.

[4] Experimental verification of the adaptive GC cycle length calculations is presented in Section 6.2.

CHAPTER 5

PRIORITIES FOR MEMORY ALLOCATION

This chapter presents a novel approach of applying priorities[1] to memory allocation and it is shown how this can be used to enhance the robustness of real-time applications. The proposed mechanisms can also be used to increase performance of systems with automatic memory management by limiting the amount of garbage collection work.

A way of introducing priorities for memory allocation in a Java system without making any changes to the syntax of the language is also proposed and this has been implemented in an experimental Java virtual machine and verified in an automatic control application.

5.1 Introduction

With the recent development in small, cheap and fast processors for embedded systems and the emerging trend of writing embedded applications in high level object oriented languages, the performance limiting bottleneck may no longer be CPU time but rather memory and memory management. This is accentuated by the high relative cost of memory in embedded systems and systems on chip.

Memory management is a system-global problem and currently puts a great responsibility on programmers. For instance, a memory leak or excessive memory allocation in one module, or component, of a system will eventually cause the entire system to run out of memory and fail. Therefore it is interesting to study whether it is possible to apply priorities to memory as well as CPU time allocation; just as we don't want an

[1]Here, we use the words "memory priority" in a sense that may correspond better to the RTSJ notion of "importance" than the real-time sense of the word priority.

important process to be delayed because a less important one is executing we don't want an unimportant memory allocation to cause a critical process to fail or be delayed, because the system runs out of memory or has to do a large amount of garbage collection work to satisfy its allocation needs.

Therefore, a novel approach is proposed which addresses two problems: firstly, how to increase program robustness by avoiding out-of-memory problems and secondly, how to increase application performance in systems with automatic memory management by reducing the garbage collection workload. Section 5.2 briefly describes both aspects, whereas the rest of the chapter will focus on the robustness issue.

While this chapter focuses on object oriented systems with garbage collection, especially Java, the robustness issues should be equally applicable to any memory allocator.

A note on terminology; in order to avoid confusion we will use the terms *high priority* (HP) and *low priority* (LP) to denote the CPU time priority of a process and the terms *critical* and *non-critical* (NC) for our new notion of priorities for memory allocations.

5.2 Applying priorities to memory allocations

It is desirable to be able to view memory allocation as any other resource allocation. The goal of this work is to provide run-time system support for doing the most important memory allocation if the system has limited memory in analogy with how the process scheduler makes sure that the most important process is run and less important ones are delayed if CPU time is scarce.

5.2.1 Avoiding out-of-memory situations

A high priority process in an embedded system may perform other tasks[2] in addition to its core functionality. For example, a digital controller process may produce log data in addition to calculating and outputting its control signal. In such a process, memory allocations by the less important tasks (e.g., producing log data) must never interfere with the core functionality (calculating the control signal).

This can be achieved by manually ensuring that the amount of log data never exceeds a certain value, e.g., by using a bounded buffer for

[2]The word task is used in the sense "a piece of work to be done" and not in the real-time programming sense. For the latter, the words process and thread are used.

delivering it to the logger process. Doing this manually has the drawback that the size of the buffer has to be calculated and this calculation is highly platform and application dependent. (I.e., each time a change that affects the application's memory allocation behaviour or the amount of memory available to the application is made, the maximum amount of non-critical memory has to be recalculated.) If more than one process does unrelated non-critical memory allocations, the complexity of managing this increases rapidly. Thus, manual solutions require a lot of work and risk being unnecessarily conservative, error prone, or both.

The proposed approach to this problem is to transfer the responsibility for making the decisions about when to allow non-critical memory allocations from the programmer to the run time system. Then, the only a priori calculation that has to be done is to calculate the amount of critical allocations done by each (high priority) process during its period and this depends only on the application and not on target platform properties like memory size.

This approach can also be used to provide a "limp home" mode, i.e., a mode of operation with lesser performance but radically lower memory consumption that will allow the application to continue executing in an out of memory situation, facilitating a more graceful degradation. This may be useful for adding some amount of predictability to applications with non-predictable memory requirements.

Finally, non-critical memory allocation gives programmers the possibility to add more features to a system without risking that these additions cause the system to run out of memory and jeopardize the core functionality of the system even if it is moved to a smaller platform. E.g., a low priority process with only non-critical memory allocations cannot cause a system to fail since, if the CPU load is dangerously high it will not get any CPU time and if the amount of memory is too low, it will not be allowed to allocate any memory.

This also has the advantage that it makes it easier to make hard real-time guarantees since worst case and schedulability analysis only has to be done on the critical parts of the system. Such analysis still has to be done using existing techniques [28, 46, 39].

5.2.2 Improving performance by reducing GC work

Another reason to limit non-critical memory allocations is to reduce the amount of garbage collection work needed and thereby increasing the amount of CPU time available to the application. This can, in turn, improve the application's performance by, e.g., allowing more advanced algorithms to be used.

Furthermore, in a real-time GC system, such as the one devised by Henriksson [22], additional memory allocations done by a high priority process may cause starvation of low priority processes; either directly, through increased execution time, or indirectly, due to the increase in GC work caused by these allocations (since the garbage collector for the high priority processes run at a higher priority than the system's low priority processes). In complex systems, however, the LP process may be more important for good system performance than a secondary task of the high priority process.

By using priorities for memory allocations, the application may be written so that, if the system runs low on memory, the primary tasks of both the HP and the LP processes are performed, but the less important task of the HP process is not. Hence, for the quality of service of the system, performance can be tuned in a more flexible and appropriate manner.

5.3 Non-critical allocations

The semi-concurrent garbage collection scheduling model introduces a special garbage collection scheduling for the high priority processes in order to guarantee that they are never delayed. Here, this is taken one step further by also considering the behaviour of the memory allocator and the risk of running out of memory, due to, for instance, unpredictable application behaviour or even wrong worst case estimates. This is done by introducing the notion of non-critical memory allocation requests, i.e., requests for memory that the run-time system may choose to deny without causing the program to fail.

Ultimately, what we want to do is to keep the amount of live non-critically allocated memory below a certain limit in order to make guarantees that critical allocations never will fail. Unfortunately, live memory amount is not a very suitable measurement, since keeping track of this is not always practically possible.

Particularly, in automatically managed memory systems, where we have the problem with floating garbage[3], there is no real way of knowing how much live memory there is in the system. The only factor we can be sure of is the amount of memory available for allocation, so we need to base our decisions on that.

[3]Floating garbage is memory that is no longer reachable from the application but has not yet been reclaimed by the garbage collector.

5.3.1 Non-critical allocation limit

The decision whether to grant or deny a non-critical memory allocation request has to be as simple as possible if it is to be used in high performance applications. That is accomplished by introducing an allocation limit for non-critical allocations; if there is less free, or ***allocatable***[4], memory than this limit, no non-critical allocations may be done. This limit will vary over time; at the start of a GC cycle, we have to reserve memory for all the (critical) HP memory allocations needed during this GC cycle and then, as the HP process runs and does its allocations, the amount of reserved memory is reduced accordingly. Figure 5.1 shows schematically how the amount of allocated, reserved and free memory varies over a GC cycle.

When deciding whether to grant or deny a non-critical memory request, we look at how much allocatable memory there is, and how much memory we need to reserve for the HP process so that all its remaining memory allocations during this GC cycle will succeed. Let n be the number of HP periods in a GC cycle, and m_{HP} the amount of critical memory allocated during each period by the HP process. Then, i HP periods into a GC cycle we need to reserve $R_{HP_i} = (n - i)\, m_{HP}$ bytes for the remaining HP periods during this GC cycle. Non-critical memory allocations should only be allowed if they won't cause the amount of allocatable memory to drop below R_{HP}.

5.3.2 Fixed GC cycle length

In order to be able to guarantee that the HP process always will get the memory it requests, we need to make sure that the GC always keeps up with the application. I.e., after each invocation of an HP process, the GC must do enough GC work so that all the allocations during the next HP process invocation will succeed. Given the amount of memory allocated by the HP process each period and the amount of memory reserved for HP allocations, we can calculate the GC cycle time expressed in number of HP process periods. We call this time the nominal GC cycle time.

To ensure that no HP allocation fails, we need to complete each GC cycle within this time, even if the actual amount of allocations done during the current GC cycle are less than the worst case. Otherwise, the

[4] Allocatable memory is memory that is immediately available for allocation. We prefer the term allocatable memory to free memory since, depending on the memory allocator or garbage collection algorithm used, the term free memory may be difficult to define or even irrelevant. E.g., in a non-compacting system, the amount of free memory may be much larger than the amount of allocatable memory due to fragmentation.

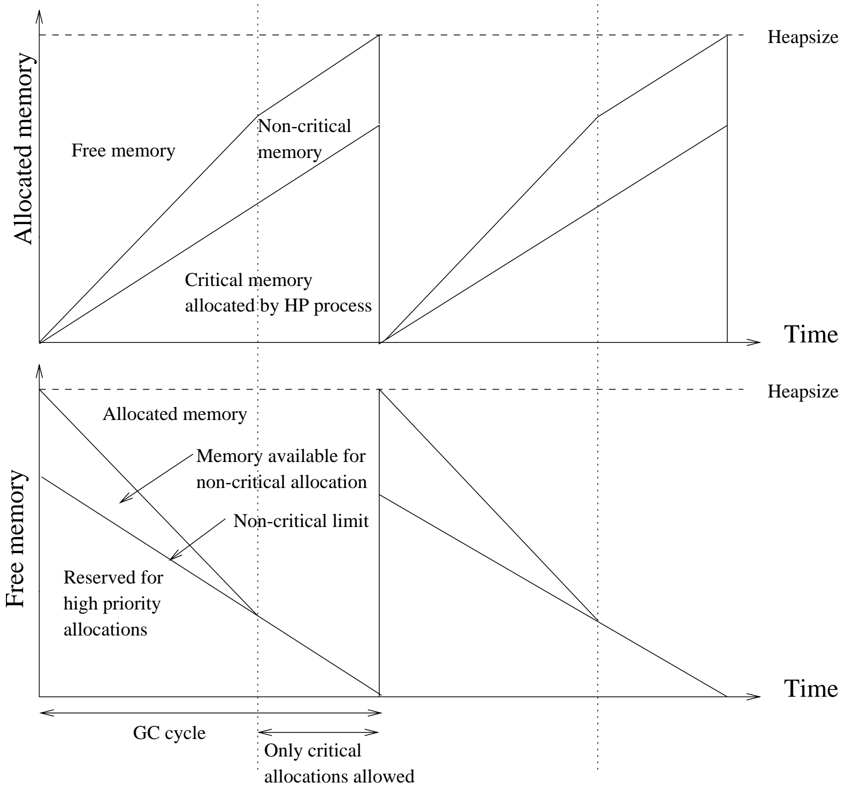

Figure 5.1: *Schematic illustration of the limit for non-critical allocations. The dotted lines indicate the times where the non-critical limit is equal to the amount of allocatable memory, i.e., when the system starts to deny non-critical allocation requests.*

situation may arise that there is allocatable memory left, but not enough for another complete HP process invocation. If a HP process is started at that time, it will require more memory than currently available and thus, that HP process will be delayed by panic garbage collection.

5.4 Detailed description

This section describes the suggested approach in more detail. We discuss how the garbage collection cycle length can be calculated, how the decisions about when to deny non-critical memory allocation requests are taken, how the scheduling can be done and finally we give an example of how such a system may work.

5.4.1 Calculating the GC cycle length

Since we want to be able to make guarantees that the application never will run out of memory while still having hard real time constraints, we need a simple model so that we can make e.g., schedulability analysis. This is done by using a fixed GC cycle time which is calculated at application design-time.

The GC cycle time, the allocation rate of the HP process and the amount of memory available for non-critical allocation all affect each other and there are several ways to calculate the cycle length. One approach is to define how much memory should be reserved for HP allocations each GC cycle, M_{HP}. If the HP process allocates m_{HP} each period we get the GC cycle length expressed in HP periods:

$$T_{GC} = n \cdot T_{HP};\ n = \frac{M_{HP}}{m_{HP}} \tag{5.1}$$

Here, the GC cycle length will be the same regardless of how much total memory the system has and changes to the amount of memory will only affect how much non-critical allocation that can be made.

Another way is to define the ratio of memory reserved for HP processes to non-critical memory. This has the advantage that the application will behave in the same way, with respect to non-critical allocations, independent of how much memory the system it's running on has. This is preferable since while non-critical allocation cannot cause an out of memory situation, they add to the amount of GC work that has to be done and thus affect the schedulability analysis. Using the ratio of critical to non-critical memory instead of a fixed amount for one of the quantities has the property that the (amortized) amount of GC work

per allocated object is independent of the total size of the memory — the memory size only affects the length of the GC cycles. Thus, this approach reduces the platform dependency of the schedulability analysis.

5.4.2 Live memory and floating garbage

In all calculations we must account for the amount of memory that lives across GC cycle boundaries and floating garbage that may exist in the worst case. This can be viewed as a reduction of the (usable) heap size with a constant. If this isn't taken into account, there will be less available memory at the start of each GC cycle than we have calculated with and the application will run out of memory.

Less obviously, it is also a problem if there is *more* allocatable memory at the start of a GC cycle than in the worst case, since this leads to the amount of memory available for non-critical allocations becoming too large, which could cause problems later. Therefore, we need to compensate for this, so that we always assume the worst case (i.e., we reserve a portion of memory to allow the amount of live memory or floating garbage to increase in the future).

With this taken into consideration, the least amount of free memory required for non-critical allocations during period i can now be expressed as

$$L_{NC_i} = (n-i)m_{HP} + f(A_{start}, C) \ ; 1 \leq i \leq n \tag{5.2}$$

where A_{start} is the amount of allocated memory at the start of this cycle, C the maximum amount of live and floating objects, and

$$f(x,y) = \begin{cases} y - x & ,x < y; \\ 0 & ,x \geq y; \end{cases} \tag{5.3}$$

5.4.3 GC for the low priority processes

When we add LP processes to the system, they will also allocate memory but the GC work corresponding to their allocations will be done at allocation time using traditional incremental GC. When LP allocations are done, the actual GC cycle time will be less than the nominal cycle time. In a traditional incremental garbage collector, this happens naturally; the extra GC work done by the LP process advances the current GC cycle.

In our system where GC work is triggered by time, however, we have to explicitly shorten the current GC cycle. Furthermore, the new, shorter

cycle time still has to be a whole number of HP process periods to ensure that there always is enough allocatable memory for one full HP process invocation. This is done by decreasing the current cycle time by l HP periods, where

$$l = \left\lceil \frac{A_{LP}}{m_{HP}} \right\rceil \tag{5.4}$$

and A_{LP} is the amount of memory allocated by the low priority processes. Thus, if the nominal GC cycle length is n HP periods, the effective GC cycle length due to LP memory allocations will be n' HP periods, where $n' = n - l$.

Note that this should only affect the effective GC cycle length (i.e., the scheduling) and not the NC limit calculations. If we were to adjust the NC limit accordingly when the GC cycle was shortened, it would be possible for non-critical allocations in a HP process to “steal” the GC work done for a critical allocation in a LP process, and that is not what we want.

On the other hand, we do need to change the NC limit due to the actual LP allocations made, because if we don’t, we would effectively reduce the amount of memory available for NC allocations. This may seem counter-intuitive but bear in mind that the purpose of the NC limit is to limit the amount of non-critical allocations and has nothing to do with controlling the critical allocations in the low priority processes.

As described above, when an allocation is made in a LP process, the corresponding GC work is done incrementally and the GC cycle is shortened so that there still will be memory for a whole number of HP process activations. Also, when a LP allocation is done, the amount of allocatable memory is decreased and in order to maintain the same amount of memory available to non critical allocations we have to reduce the NC limit with the same amount as the size of the LP allocation.

If we have allocated A_{LP} bytes of memory in the LP processes during this GC cycle, the NC limit can be written

$$L_{NC_i} = (n - i)m_{HP} + f(A_{start}, C) - A_{LP} \tag{5.5}$$

5.4.4 Non-critical limit calculations in the real world.

In all the previous calculations in this chapter, we have assumed that a GC cycle can easily be divided into a number of HP process periods and that the memory allocations of each period are done instantaneously at the start of the period. This model is well suited for reasoning about systems and off-line analysis but doesn’t lend itself well to actual implementation.

In real systems, the high priority processes often have different period times, and real programs do allocations more or less sporadically during their execution rather than at the start of a well defined period. For these reasons, among others, a NC limit based on the number of elapsed HP periods is not a very practical one for run-time calculations. Instead, we will use the following algorithm:

- At the start of each GC cycle, the amount of memory needed by all the critical allocations by HP processes is calculated[5]. This is the amount of memory reserved for HP allocations (compensated for floating garbage, etc), $R_{HP} = M_{HP} + f(A_{start}, C)$
- Whenever a critical HP allocation is done, R_{HP} is decreased by the size of the allocated object. When an allocation is done by a low priority process, A_{LP} is increased. The non-critical limit is then updated; $L_{NC} = R_{HP} - A_{LP}$.
- If the amount of allocatable memory is less than or equal to L_{NC}, non-critical allocation requests will be denied.

This way, the NC limit will always be correct, regardless of how much memory the HP processes actually allocates and at what time during their execution they perform the allocations.

Another implementation issue is that our calculations assume that the garbage collector only frees memory at the very end of each GC cycle. This simplifies the non-critical limit calculations as each cycle can be viewed independently but when implementing support for non-critical allocations, care must be taken to assure that this assumption holds.

Mark-sweep collectors, of course needs some attention as they, by nature, free memory continuously during the sweep phase. A copying collector has this behaviour in principle, but still might have to be modified; it does free all memory after the last object has been moved, but this could happen before the full GC cycle time has elapsed.

Thus, in any case the memory manager must be designed so that it does not make any memory available to the allocator until at the start of the next GC cycle. Otherwise, too many non-critical allocations might be allowed in the current cycle, which might cause problems later. This also means that if the GC work metric is conservative and the garbage collector finishes early, the freed memory should not be made available to the allocator until at the start of the next cycle.

[5] The actual calculation of the worst-case memory requirements for each process could be done either manually or at compile time. Another possibility for soft real time systems is that it could be estimated by the run-time system based on measurements from previous GC cycles.

5.4.5 Time-based GC scheduling

Traditionally, incremental garbage collectors have been implemented so that GC work has been triggered by memory allocation, and done in proportion to the amount of allocated memory. I.e., when half of the memory available at the start of the cycle has been allocated, half of the GC work required to complete the cycle has been done and when all the memory has been allocated the GC cycle is completed.

That approach to GC scheduling is does not fit well into a system with non-critical allocations. The problem is that it may cause low memory utilization; If the application does less critical allocations than its worst case the GC cycle will be longer. The limit for non-critical allocations, on the other hand, is not affected, so when the amount of allocatable memory reaches the non-critical limit, no more non-critical allocations are allowed during that GC cycle. Thus, the less critical memory the application allocates, the longer the GC cycle gets and the less non-critical allocations are allowed, which is not what we want.

Therefore, we use time, rather than allocation, as the trigger for GC work and do GC work in proportion to how large a fraction of the GC cycle time has elapsed. I.e., when half of the GC cycle time has elapsed, the GC should have done (at least) half the work needed to complete the cycle. This ensures that each GC cycle finishes within the fixed time, even if there is allocatable memory left. Thus, time-triggered GC ensures the same non-critical memory behaviour regardless of how much critical memory the application actually allocates (as long — of course — as the allocated amount is less than the assumed worst case).

5.4.6 Example

As an example, we take a system with one high priority process doing both critical and non-critical memory allocations and a set of low priority processes doing critical memory allocations.

In figure 5.2 you see how the amount of allocated and allocatable memory, respectively, varies over three GC cycles. In the first GC cycle, the amount of memory reserved for critical HP allocations (or rather, the non-critical limit) is larger than in the other two. This is because we must compensate for the fact that there is *less* than the maximum amount of allocated memory at the start of the GC cycle (see Section 5.4.2).

The second GC cycle shows how the system behaves when there are no allocations (and thus no incremental GC work) done by the low priority process. The first and third cycles are shorter than the nominal cycle length since low priority allocations are done.

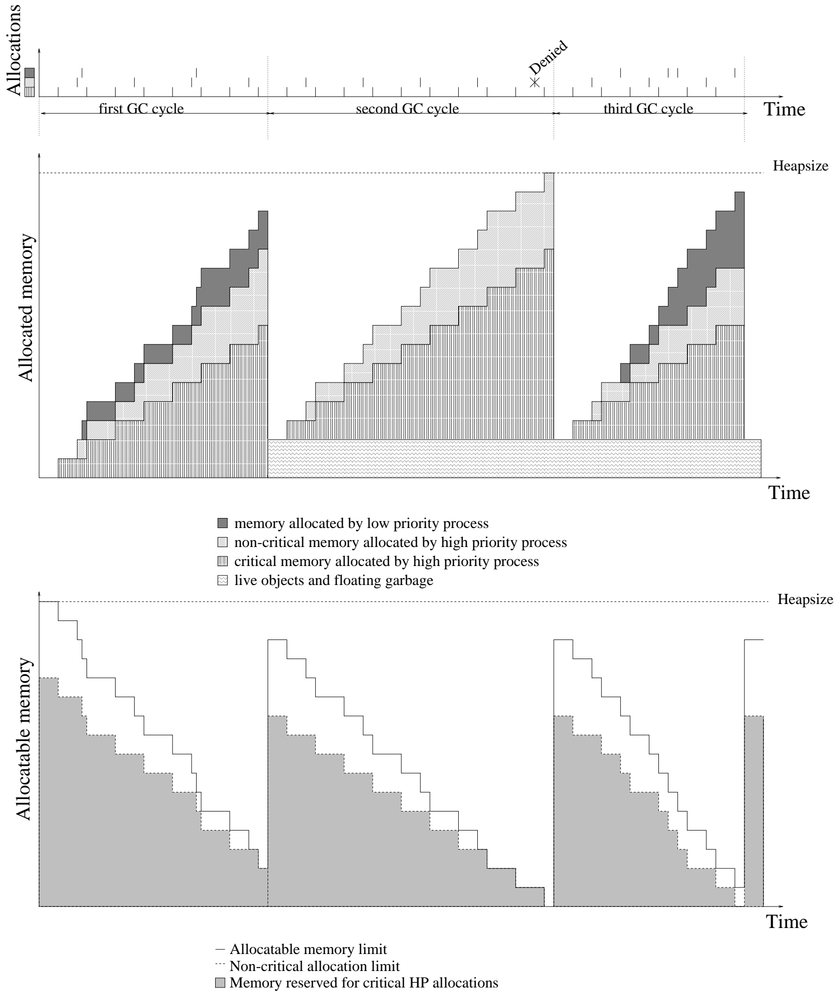

Figure 5.2: *An example showing how the amounts of allocated and allocatable memory vary over time. Allocation requests for non-critical memory are denied when the amount of allocatable memory is less than or equal to the non-critical allocation limit ($R_{HP} - A_{LP}$). This happens at the end of the second GC cycle. Note that the first and third GC cycles are shorter than the nominal length due to low priority memory allocations. Also note how the non-critical limit is lowered when LP allocations are done so that the amount of memory available for non-critical allocations is not changed.*

Since we have a fixed nominal GC cycle length and use time, rather than memory allocation, to trigger GC work the GC cycles may end before all available memory has been allocated. This can happen if the application uses less memory than in the worst case or due to quantization when low priority allocations are made (see section 5.4.3).

5.5 Non-critical memory in Java

The main objective when implementing these ideas in a Java environment was that no changes to the syntax of the Java language should be made, and that programs written for our system should work on any Java platform (but, of course, without the added semantics of non-critical memory allocations).

The proposed approach is to use the exception mechanism of Java, so we define an exception class, `NoNonCriticalMemoryException`, with the added special semantics that all allocations that are done in a block which catches that exception are non-critical. Figure 5.3 shows a simple program which does both critical and non-critical memory allocations. This program will run on any Java platform with the only addition of an (empty) exception class.

```
1  void example(){
2      Object aCriticalObject = new Object();
3      foo(aCriticalObject); // do something important
4      try{
5          Object aNonCriticalObject = new Object();
6          foo(aNonCriticalObject);
7          doSomething();
8              // do something
9              // if the non-critical
10             // allocation was successful
11         } catch(NoNonCriticalMemoryException e){
12             // non-critical allocation failed
13         }
14 }
```

Figure 5.3: *Small example program. The allocation of `aCriticalObject` is always done, but the allocation of `aNonCriticalObject` may be denied. If the allocation fails, a `NoNonCriticalMemoryException` is thrown and may be handled in the catch-clause.*

Non-criticality is transitive, i.e., memory allocations in a method that is called from a non-critical region, like the calls to the methods `foo()` and `doSomething()` on lines 6 and 7 in Figure 5.3, are also non-critical. Note, however, that the first call to `foo()`, on line 3, is *not* non-critical since the call is not made from a non-critical block. This behaviour is preferable since an auxiliary function could be called both from critical and non-critical contexts. In order to make such transitivity possible without having to litter the code with `try` and `catch` clauses, the exception class `NoNonCriticalMemoryException` is an unchecked exception. An instance of this class can be statically allocated to avoid wasting memory.

An experimental implementation has been made using the IVM (Infinitesimal Virtual Machine) [23], a very compact real-time Java virtual machine. Currently, non-critical allocations are explicitly turned on and off using a native method `IVM.setMemoryPriority()`. This is, however, not fundamentally different from our proposed approach since the `setMemoryPriority()` calls could be inserted automatically by the class loader as the exception table is set up (much in the same way as `monitorenter` and `monitorexit` byte codes are inserted for synchronized methods).

5.6 Summary

The idea of applying priorities to memory allocation was introduced and it was shown shown how this can be used to enhance the robustness of real-time applications. The advantage this approach gives is twofold:

Firstly, it provides run-time support for prioritizing memory allocations if there is not enough available memory to safely accommodate for all allocation requests. Secondly, but equally important, it makes it easier to provide hard guarantees since the worst case memory usage calculations only has to be done for the critical parts of the system as non-critical allocations cannot cause the system to fail. Furthermore, it is suggested that the same mechanisms could be used to increase performance by limiting the amount of memory allocation and, consequentially, GC work.

It is observed that memory priority and CPU time priority needs to be treated separately. The logging example shows that a process having high CPU time priority doesn't necessarily mean that all of its memory allocations are critical.

The presented approach is based on the notion of non-critical memory allocation requests, which can be used by the programmer to indi-

cate that the memory allocations done in a certain part of the program are less important than the rest. Such non-critical allocations may be allowed to fail if the run-time system decides that that memory could be of better use elsewhere or that the increased garbage collection work would degrade system performance.

The incorporation of priorities for memory allocations in an object oriented language is studied and a way of introducing non-critical memory allocation in a Java system without making any changes to the syntax of the Java language is proposed. This has successfully been implemented in the IVM experimental Java virtual machine.

Preliminary experiments[6] show that the mechanism is fairly easy to implement and can improve the robustness and performance of a control application by restricting its operation to the critical tasks if the system runs low on memory. It allows the programmer to write a system that performs better if run on a faster and larger system but whose critical tasks won't fail if it is run on a system with less than ideal amount of memory. Instead, the non-critical features of the system will automatically be turned off if there isn't enough memory for them to be safely executed.

[6]The experiments with priorities for memory allocations are presented in Section 6.3.

CHAPTER 6
EXPERIMENTAL VERIFICATION

This chapter presents experimental support for the proposed techniques. As a test platform, a simple control system for a lab process which balances a ball on a beam was used. The angular velocity of the beam is controlled in order to roll the ball to a given position on the beam. A photo of the lab process is shown in Figure 6.1.

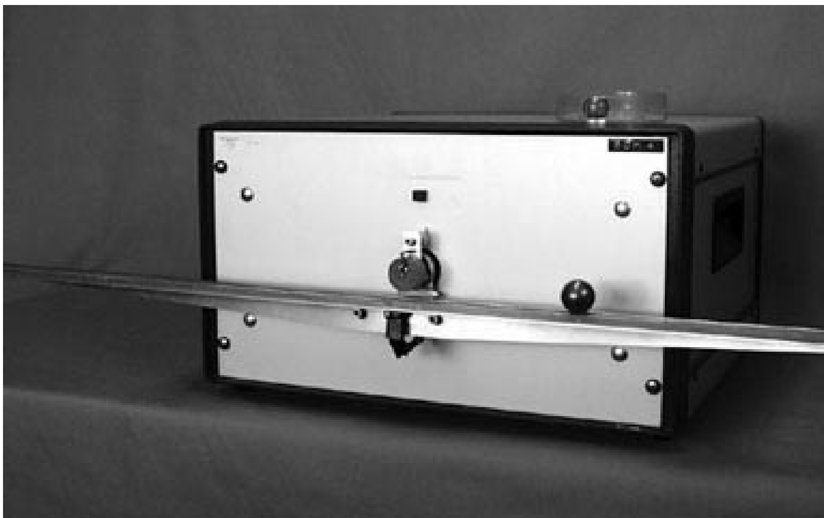

Figure 6.1: *The ball-on-beam process. The beam can be rotated to roll the ball to the desired position. Sensors measure the position of the ball.*

The control was performed by a Java application consisting of three threads; a user interface, a reference generator, and a controller. In addition to doing the actual control, the controller thread sends log data back to the user interface thread as illustrated in Figure 6.2. The reference generator and controller are run at a much higher rate than the UI thread.

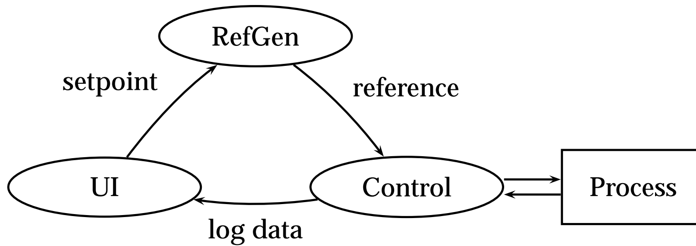

Figure 6.2: *The ball-on-beam control application consists of three threads; user interface, reference generator and controller. The data communicated between the threads is indicated by the arrows.*

In the plots showing the thread scheduling, the threads are numbered as follows: idle (-2), GC (-1), main (0), controller (1), reference generator (2) and UI (3).

The garbage collector used is an incremental mark-compact collector. The traces were collected by instrumenting the RT-kernel and the Java virtual machine, respectively, with logging calls at memory operations and context switches. Logging was done to a dedicated memory area and uploaded via a serial line after each experiment. The time-triggered and adaptive GC experiments were performed using compiled Java [37] on a 350 MHz PowerPC and the memory allocation priority experiments were done using the IVM virtual machine [23] on a STORK [3]/Linux platform.

6.1 Time-triggered GC

This section illustrates the run-time behaviour of allocation-triggered and time-triggered garbage collection and shows the difference between traditionally scheduled incremental GC, where each increment is scheduled individually and the work is spread evenly across the GC cycle, and EDF-scheduled time-triggered GC.

Figure 6.3 shows an execution trace of a run with allocation triggered increments, in Figure 6.4 the same program is run with time-triggered GC with metric-scheduled increments and Figure 6.5 shows the corresponding trace with time-triggered, EDF scheduled garbage collection. At the macro level, the executions are almost the same; the memory traces are nearly identical and the mutator threads get to run when they should. The big difference is between the versions where the individual increments are scheduled separately, in order to spread the work evenly across the cycle, and the EDF-scheduled version. Figure 6.6 and Figure 6.7 show a close-up view of the thread graphs. Here, you can see that the allocation-driven garbage collector performs a much larger number of miniscule increments as it spreads the GC work more evenly across the GC cycle even though there is idle time in the schedule. The deadline-scheduled version, on the other hand, finishes as quickly as possible, which is shown by the longer GC invocation without any idle time.

If the application has a bursty allocation pattern, the difference between allocation- and time-triggered scheduling gets more discernible. A simple experiment where the low frequency UI thread was modified to allocate a large number of objects at each invocation was performed. Memory traces of this execution is shown in Figure 6.8 and Figure 6.9, and close-ups of the thread graph is shown in Figure 6.10 and Figure 6.11. In this case, both the memory trace and the scheduling are different.

The difference between allocation-triggered and time-triggered GC when it comes to handling bursty allocations is shown in the scheduling graphs. When the UI thread (number 3) has executed and made the large allocation, the following GC increment is much longer than the other increments. As the GC locks the heap when running, the controller thread (number 2) gets blocked by the GC thread, causing jitter. Also notice that, by necessity, the cycle length of the time-triggered GC has been shortened in order to accommodate the higher allocation rate.

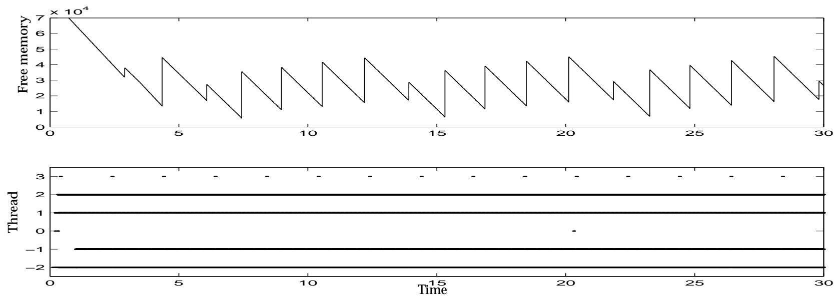

Figure 6.3: *Memory trace and schedule for the ball on beam application using allocation-triggered GC.*

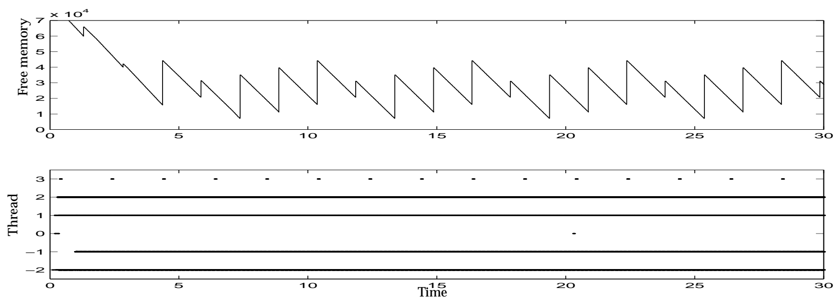

Figure 6.4: *Time-triggered with individually scheduled increments.*

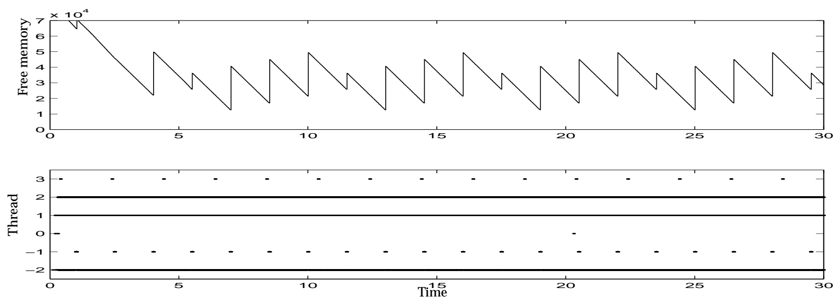

Figure 6.5: *Time-triggered, EDF scheduled.*

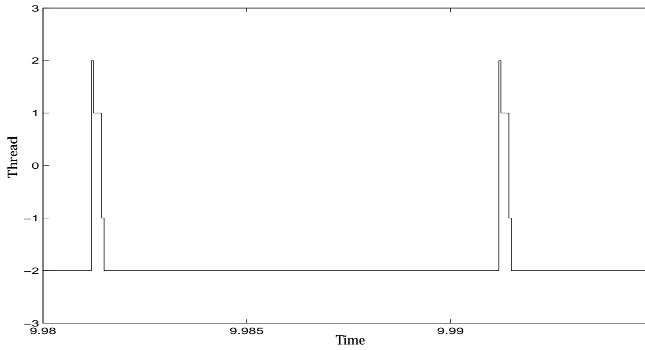

Figure 6.6: *Thread scheduling with the allocation-triggered GC. As the allocations performed during each thread period is small, the corresponding GC increment is also very short. The schedule of the time-triggered, metric-based scheduler is quite similar as both schedulers spread the GC work evenly across the cycle and the constant allocation rate of the application makes it possible to tune the work metric used in the allocation-triggered GC.*

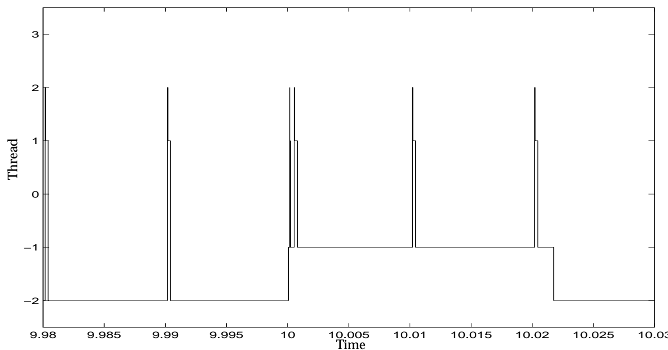

Figure 6.7: *Thread plot with the EDF scheduled GC. When a GC cycle is started, the garbage collector uses all idle time in order to perform the work required to finish the GC cycle as quickly as possible and then remains idle until the start of the next cycle. Each increment is, however, still very short in order to avoid disturbing the application threads more than necessary. This can be seen at* $t = 10\ s$. *Here, the GC thread is released just before the application threads. Thread number 2 preempts the GC, but since the GC has locked the heap, when thread 2 attempts a heap operation it is blocked until the GC finishes its current increment. Thread 2 was blocked for* 0.4 *milliseconds.*

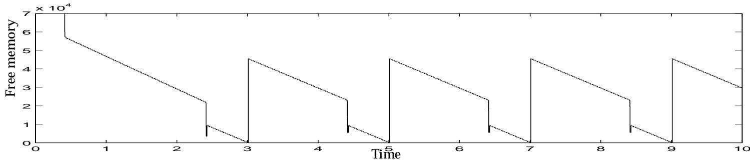

Figure 6.8: *Memory trace of an application with bursty allocations and allocation-triggered GC.*

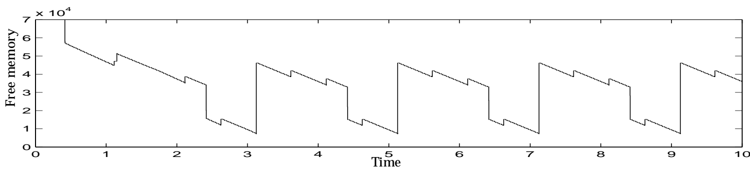

Figure 6.9: *Memory trace of an application with bursty allocations and time-triggered GC.*

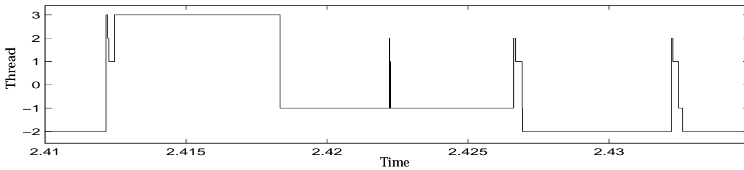

Figure 6.10: *Part of the thread graph corresponding to Figure 6.8. Note how a large allocation in thread 3 causes a long GC increment.*

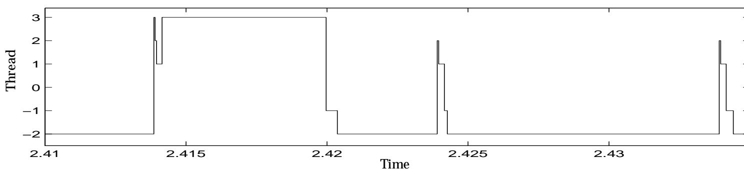

Figure 6.11: *Part of the thread graph corresponding to Figure 6.9. As GC work is not triggered by allocations, the GC work is spread evenly across the GC cycle, and long increments are avoided.*

6.2 GC cycle time auto-tuning

This section examines the adaptive GC cycle time estimates described in Section 4.3. In the experiments presented in this section, EDF scheduling was used for the GC thread.

Figure 6.12 shows a memory trace of the system with the auto-tuner enabled. The fast threads run at 100 *Hz*. Figure 6.13 shows how the auto-tuner reacts to changes in allocation rate. At $t = 10\ s$, the frequency of the high priority threads is increased from 20 to 100 *Hz* and at $t = 20\ s$ the frequency is lowered to 20 *Hz*. The GC is scheduled so that it will work even if all the dead objects in one cycle would be floating garbage. I.e., we reserve a part of the available memory for the next GC cycle as expressed in Equation (4.5).

As memory allocations typically are bursty, the measurement of the allocation rate is filtered in order to keep the deadline estimates more stable and reduce the update frequency for the scheduling parameters. Care must be taken not to underestimate the allocation rate, as this might lead to an out-of-memory situation, so we must react quickly to actual changes in allocation rate while avoiding chatter due to bursty allocations. The rise time in the allocation rate plots are due to such filtering.

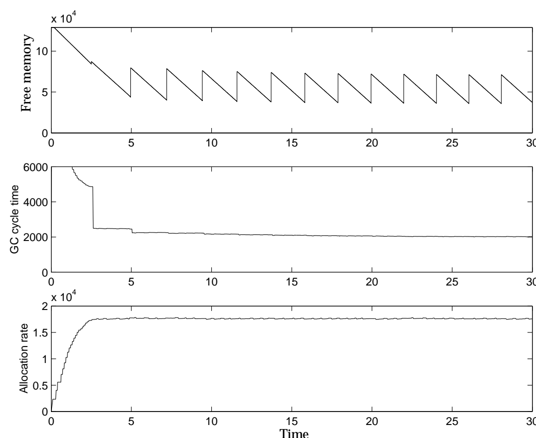

Figure 6.12: *Memory trace of the system with adaptive GC cycle length. The topmost plot shows the amount of available memory (in bytes), the middle plot shows the estimated GC cycle length (in milliseconds) and the bottom plot shows the LP filtered allocation rate measurement (in bytes/second).*

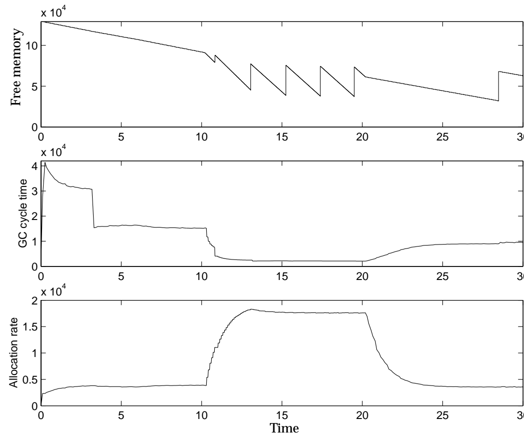

Figure 6.13: *How the GC scheduler reacts to changes in allocation rate; At $t = 10\ s$, the frequency of the high priority threads is increased from* 20 *to* 100 *Hz and at* $t = 20\ s$ *the frequency is lowered to* 20 *Hz.*

6.3 Priorities for memory allocation

It was claimed that introducing priorities for memory allocations and run-time system support for denying unimportant memory allocations if memory is scarce can help increasing both the robustness (by avoiding out-of-memory situations) and performance (by limiting the amount of garbage collection work) of real-time systems. This section presents experimental support for those claims.

6.3.1 Avoiding out-of-memory situations

Two scenarios where non-critical memory allocations can help making sure that a change to a previously working system doesn't risk breaking it was encountered: increasing the sampling rate of the controller and reducing the amount of memory available to the application.

When the sampling rate is increased, the controller both uses a larger part of the CPU time and allocates log data at a higher rate until we get to a point where the user interface thread doesn't get the CPU time needed for consuming all the log data and the application runs out of memory and fails. By making the log data allocations non-critical, this cannot happen and the control is not affected.

Reducing the available memory[1] will, obviously, at some point cause the application to fail. However, by making the allocation of log data non-critical, the minimum memory requirement for the application may be significantly reduced compared to the original version.

The following traces illustrate the first scenario. In these experiments, the period of the reference generator and the controller was both 20 ms, and a log data object about 60 bytes. Figure 6.14 shows a run of the ball-on-beam system without non-critical memory. The high allocation rate causes a large GC workload and the UI process is starved, eventually leading to failure.

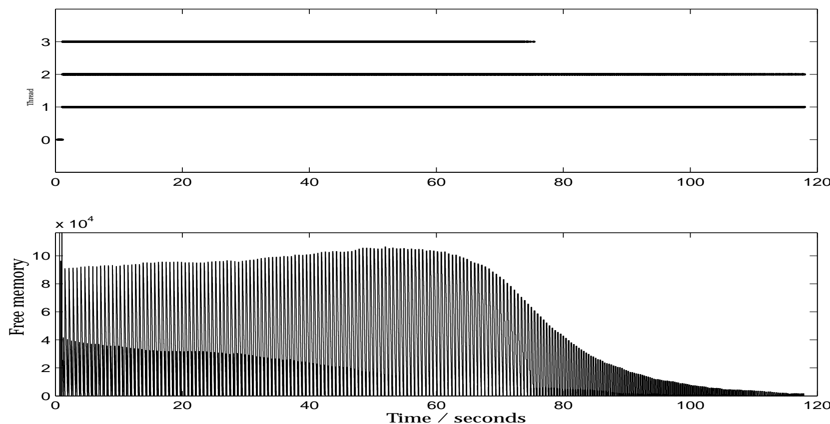

Figure 6.14: *A sample run of the ball-on-beam system without non-critical memory. The UI thread* (3) *doesn't get enough CPU time to consume all plot data that is produced. After* $t = 75$ *it is totally starved by the GC. Then, less and less memory is available and more and more CPU time is spent doing panic GC.*

In the first half of the run the controller*(1)* and reference generator*(2)* threads run unimpeded, and the control was OK until $t = 90$. After that the frequent panic stop-the-world GC cycles caused so long delays that the controller dropped the ball. The CPU load is almost 100% and the idle thread *(0)* is not run except in the very beginning. The reason that the maximum amount of allocatable memory increases in the middle is that when the GC cycles get shorter there is less floating garbage.

Figure 6.15 shows the same system where the allocation of log data has been made non-critical, and the log data allocation is kept at a sus-

[1]This could occur either by actually running the system on a smaller platform or, perhaps more likely, by adding more threads to the system.

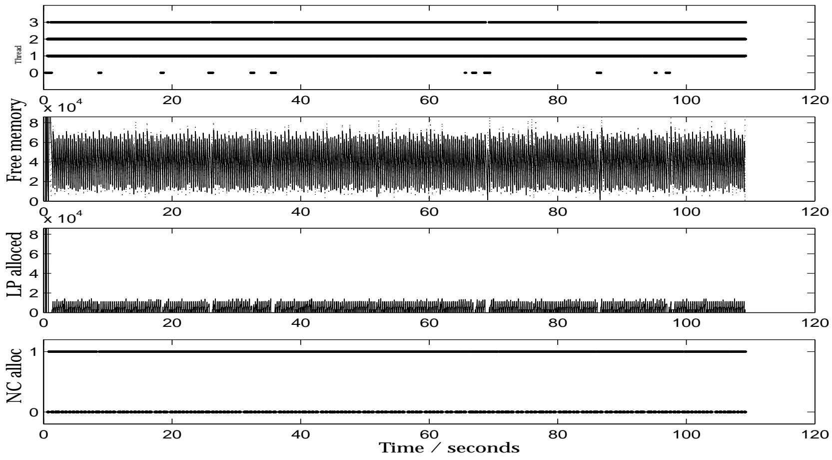

Figure 6.15: *A run of the ball-on-beam system with log-data allocations made non-critical. In the thread plot you see that the UI thread gets CPU time throughout the run. The third plot shows the amount of memory allocated by low priority processes during this cycle. The fourth plot shows if non-critical allocations succeed or not; high level means success and low level is deny.*

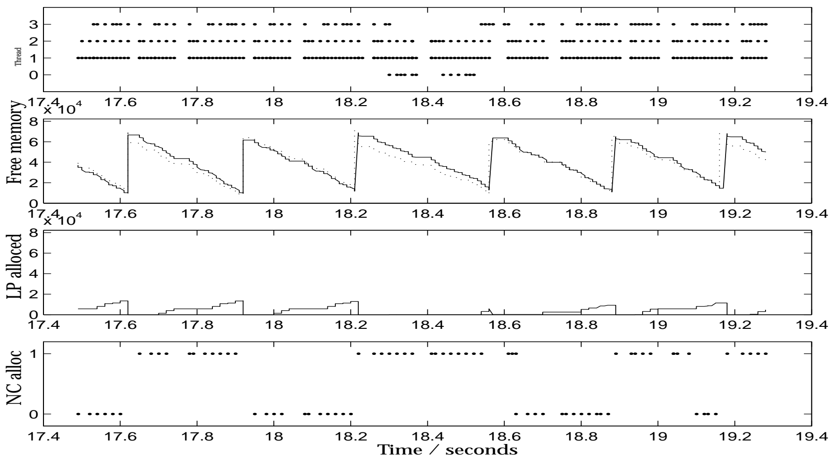

Figure 6.16: *Close-up to show the non-critical memory behaviour. The dotted line in the free memory plot is the non-critical limit. Note how the GC cycles are shortened when low priority allocations are made.*

tainable level. In this experiment, more than half of the log data allocation requests were allowed. Figure 6.16 shows a close-up of Figure 6.15 where you can see the non-critical behaviour more clearly.

6.3.2 Improving performance

The experiments also indicate that it is possible to achieve better control performance by limiting the amount of non-critical memory allocations. The plots in Figure 6.17 show two runs of the ball-on-beam application without and with non-critical memory allocations enabled, respectively. The position of the ball is in the interval $[-10, 10]$.

In the version without non-critical allocations, the high allocation rate occasionally forces the garbage collector to do a full garbage collection cycle in order to reclaim enough memory to satisfy the allocation needs. This delays the high priority controller process so that it misses its deadline which, in turn, degrades the control performance.

When the allocation of log data is made non-critical, the allocation is kept below the safe limit and the system runs as designed, with more consistent control performance.

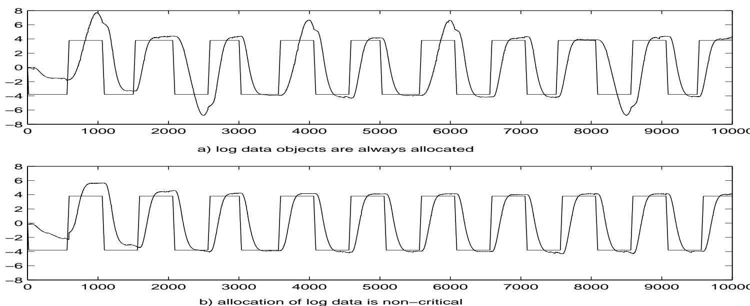

Figure 6.17: *Plots showing the reference value and the measured position for the ball-on-beam process. Plot* a *shows the system without non-critical memory allocations and plot* b *shows the system where the allocation of plot data is non-critical. The irregular behaviour in* a*, around samples 2500, 4000, 6000, and 8500, is caused by the controller process being delayed by the garbage collector due to the program running out of allocatable memory and forcing a complete garbage collection cycle.*

CHAPTER 7

FUTURE WORK

The results presented in this thesis are preliminary and many open problems remain to be examined, mainly in the areas of adaptive GC scheduling and priorities for memory management. Proof-of-concept implementations which supports the feasibility of the respective techniques have been presented, but a prototype which integrates all of the ideas of this thesis in a feedback scheduling system should be implemented in order to study the synergy effects that could be obtained. This chapter outlines our plans for future research.

7.1 Adaptive GC scheduling

The auto-tuning of the garbage collection cycle length presented in this thesis uses a black-box approach; we do not require any knowledge of the internals of the garbage collector or mutator — the only quantity that is measured is the amount of available memory. This has the advantage that it is very easy to plug this kind of auto-tuner into an existing system or to change garbage collectors, as very little communication with the memory subsystem, and none at all with the mutator, is required. The drawback is of course that we cannot react to changes in allocation rate until they occur.

A more sophisticated model would take knowledge about the application program into account. For instance, if some threads are periodic, we could take advantage of the knowledge that each thread does a certain amount of allocation during each invocation and then is idle until its next invocation. Then we could measure and estimate how much each thread allocates during each invocation which might even further

mitigate the problems with apparently bursty and random allocation patterns. Another improvement could be to use information about the execution patterns of the threads, for instance through feed-forward of changed sampling periods, etc.

The experiments presented are of a preliminary nature and the performance requirements of the application were modest. In the near future, we plan to evaluate the real-time performance of our time-triggered garbage collector prototype under more challenging conditions in a high performance robotics application. The execution time estimation discussed in Section 4.4 requires more work and must be implemented in a real run-time system.

Another interesting research issue is raised by the difference in how the GC increments are scheduled in the fixed priority and EDF systems described in this thesis: Is it desirable to spread GC work evenly across the cycle even if that means leaving idle time at the start of the cycle? One advantage of that approach is that it may give objects allocated at the start of the cycle time to die, which decreases the average amount of floating garbage. The major drawback is that it leaves less slack in the schedule towards the end of the cycle and therefore makes the system more vulnerable to changes in CPU utilization. This may be of particular importance in an adaptive system where robustness to variation in resource utilization is one of the key factors.

7.2 Priorities for memory allocation

Preliminary experiments indicate that having run-time support for dividing memory allocations into critical or non-critical can increase both robustness and performance of real-time software. However, more experiments on larger systems and systems with high performance requirements (e.g. low latency) will have to be done.

It would also be interesting to study whether additional advantages may be gained from having an arbitrary number of memory priority levels compared to having just two (critical and non-critical).

7.2.1 Configurable behaviour

Models for controlling when to fail non-critical allocations should be studied. In the logging example the optimal behaviour of the system depends on what the intended use of the log data is; if it is for system identification we want as long consecutive series of data as possible but the amount of time between the series is of less importance. Therefore,

in such an application, we want every non-critical allocation request to be granted up to a point where no more non-critical requests are granted during that cycle. On the other hand, if the data is to be used for plotting, we want the samples to be equally spaced, i.e., every nth non-critical allocation request should be granted. Furthermore, usually a set of allocations is needed in order to perform a certain task. If the last allocation of such a set is denied, the whole task has to be abandoned for that time. That should also be taken into account when deciding whether to grant or deny an allocation request.

Also, would it be possible to have different profiles to let the programmer choose among to get the one that fits a particular application best? Could such profiles co-exist in one application, i.e., different parts of the application having different non-critical memory policies?

7.2.2 Non-critical memory using aspects

In this work, focus is on embedded real-time systems and the approach, as presented here, relies on the fact that we can modify the memory allocator. For systems without hard real-time constraints, however, it may be possible to achieve the same advantages without having to do any modifications to the Java platform. One way of doing this could be by using aspect oriented software development[4]. The cross-cutting concern in this case is the handling of low-on-memory situations. It should be investigated whether it is suitable to e.g. divide the tasks into critical and non-critical aspects and dynamically weave in the non-critical parts only if the system has enough memory. We believe that it is possible to use e.g., the property-based cross-cutting of AspectJ [29] to insert a test whether an allocation should be done before each call to a constructor.

7.3 GC scheduling interface

The experimental platforms were implemented using the garbage collection interface (GCI) [24] developed by our research group. The GCI is a programmer's interface consisting of a well-defined set of memory operations and the goal of the GCI is to make it possible to separate the GC implementation from its usage even in a hard real-time system and in an uncooperative environment like an optimizing compiler back end that is unaware of garbage collection. The GCI makes it possible to change GC algorithms without making any changes to the rest of the run-time system or the code generation.

This scheduling principles presented in this thesis makes it possible to separate the GC scheduling from the GC implementation. As a black box approach to GC cycle time estimation is used in the current prototype implementation it is possible to change garbage collector without modifying the scheduler. However, if we want to allow a clear box approach, it is necessary to specify a GC scheduling interface that defines how the communication between the GC algorithm and the GC scheduler is done and that requires further investigation. Furthermore, the communication between the process scheduler and the GC scheduler must be studied and formalized.

7.4 Feedback scheduling and QoS

We believe that the ideas presented in this thesis make it easier to handle memory management issues in the feedback scheduling and quality-of-service (QoS) areas. For instance, by treating memory allocations just like any other resource allocation, it is possible to optimize the trade-off between memory usage and CPU time. I.e., increasing the allocation rate increases the GC workload which, in turn, reduces the application's CPU time and vice versa. It should be studied how this can be used in practise to optimize quality-of-service.

In a feedback scheduling based system, like the one described in [12] and [13], bringing the memory system into the loop coupled with the possibility to dynamically change an application's memory allocation behaviour depending on the system's current memory status could be used to get better overall performance. This is a very complex matter and strategies for when to deny non-critical memory allocations in order to optimize performance as well as adaptive GC scheduling strategies need further studies.

7.5 Distributed hard real-time systems

Another area where the presented techniques may have impact are temporally predictable distributed systems. In a distributed system, the nodes can be seen as components and the whole system as being constructed by composition of node components. When designing such systems, one important factor is the ease of composing systems out of components, *composability* . The time-triggered architecture [31, 32] addresses the composability problem and important features of that model are time-triggered communication and temporal firewalls — interfaces

between the components specifying what data should be available or communicated at what time. Such interfaces makes it possible to guarantee that if the individual components conform to their specified interfaces, the resulting system will work as intended. They also solve problems of safety critical systems like, for instance, maintaining a global time base and determining data validity.

In order utilize automatic memory management in such temporally predictable components, it seems as it would be helpful, if not necessary, to be able to guarantee that also the memory manager is temporally predictable. As time-triggered GC scheduling has the property that it has an explicit deadline and therefore makes it possible to guarantee that a GC cycle finishes and makes a certain amount of memory available at a certain time, it would be interesting to study the impact of time-triggered GC in this field of application.

CHAPTER 8

RELATED WORK

This chapter presents related work in the areas of GC scheduling, memory management for real-time Java, and worst case analysis.

8.1 Time-based garbage collection scheduling

Henriksson

Using time as the GC work metric was discussed in [22] as this would solve the problem of traditional GC work metrics failing to capture the temporal behaviour of the garbage collector. The approach was, however, dismissed as impractical, since it requires a high resolution clock. However, most current embedded platforms (even smaller ones, such as the Atmel AVR) have timers with resolution of the same magnitude as the CPU clock, which is more than adequate for these purposes. Thus, we believe that using time as the fundamental garbage collection work metric both offers advantages over ad hoc metrics and is practically possible.

Bacon et al

The problems of allocation-triggered GC scheduling in real-time systems, particularly the uneven GC overhead and consequentially, mutator CPU utilization, caused by variances in allocation rate, are addressed by David F. Bacon et al in a recent paper [6]. To achieve even and predictable mutator CPU utilization, time-based scheduling, where the collector and mutator are interleaved using fixed time quanta, is proposed.

The work of Bacon et al is largely motivated by the same concerns and has much in common with the work presented in this thesis. One fundamental feature of time-based GC scheduling common to both approaches is that they turn garbage collection into a periodic activity instead of a sporadic one as allocation-triggered GC does.

The main difference between the model proposed by Bacon et al and the time-triggered GC scheduling model presented in this thesis lies in the level at which GC scheduling is considered; the period time of their model is at the quantum level while the period of the time-triggered GC is the GC cycle. Also, the fixed time quanta of [6] explicitly state how the GC work should be scheduled while the time-triggered model specifies a deadline and leaves the actual scheduling decisions to the underlying process scheduler.

The behaviour of the approach of Bacon et al is, at a large time scale, similar to that of a semi-concurrent GC or a time-triggered GC in that the CPU utilization of the mutator is predictable and consistent and independent of bursty allocation rate of the mutator. [1] However, at a more fine-grained level, the garbage collector may still preempt the mutator as the GC is scheduled to run for one GC quantum after each mutator quantum. Here, the design goals behind their collector differ from the ones driving the work in this thesis; they focus on low overhead and consistent utilization while non-intrusiveness and low GC induced latency and jitter are the key issues behind this thesis.

Qian et al

Time-triggered GC was also proposed in [52] as a means to spread GC work more evenly and minimize the number of GC invocations and heap usage when the application's allocation pattern is bursty. The focus on that work is on measuring object lifetimes but they note that similar concerns are relevant in server applications.

Previous object life span studies have used an allocation-triggered approach, calling the GC every n KB of allocation. Qian et al supplement this with a time based approach by periodically performing a GC cycle, e.g., every 100 ms. In their paper, no effort is made to ensure that the collector keeps up with the mutator since this is not a problem in their application; it is sufficient that the GC cycle time can be manually tuned to suit a particular application.

[1] The interleaving of GC and background processes in the semi-concurrent model may be almost identical; quantization effects due to atomic GC primitives make a GC scheduled according to Equation (3.19) behave as a time-based GC with small GC and mutator quanta.

They also hint that the time-triggered approach can be applicable to embedded systems by using the timing information of the processes to run the GC when the number of live objects is small. The focus is still on efficiency and minimizing the number of GC invocations and they do not address any real-time issues.

8.2 Adaptive GC scheduling

An approach to adaptive GC scheduling aimed at minimizing the GC overhead is suggested by Henriksson [21]. The idea is that at the start of the GC cycle, garbage collection is performed at a rate that will allow the GC to finish on time in the average case. Then, at a certain (a priori calculated) point, if the GC workload in the current cycle was more than the average, the GC rate is increased to the maximum rate in order to finish on time. Thus, this adaptive GC rate improves the average performance while still guaranteeing that the GC will not stop the application from meeting its deadlines in the worst case.

Engelstad and Vandendorpe [19] mention using a heuristic for controlling the “steal rate” of their garbage collector. A GC increment is performed every n allocations and GC progress is measured. If forward progress is not made, n is decreased and vice versa.

Siebert [48] also use use an adaptive scheme to minimize GC overhead; based on the current memory utilization, a proper value for how much GC work to be performed for every allocated byte is determined. The fundamental difference between that work and the adaptive scheduling presented in this thesis is that Siebert requires an upper bound on the fraction of allocated memory to be known and the adaptivity is an optimization to avoid unnecessarily long GC increments if the actual amount of allocated memory is less than the worst case. The adaptive scheduling presented in this thesis requires no a priori analysis and is purely based on measuring the state of the memory system. This gives increased flexibility at the cost of a priori guarantees.

8.3 Memory Management in Real-Time Java

There are two specifications for real-time Java; The Real-Time Specification for Java (RTSJ) [10] and the Real-Time Core Extensions (RTCE) [25]. Both try to solve the real-time garbage collection problem by avoiding it. They assume that garbage collection is not feasible in real-time systems and instead propose region-based approaches to memory management

for the real-time threads. The non-real-time threads do their memory allocation on a heap with traditional garbage collection.

RTSJ uses *scoped memory areas* for high priority threads. Objects allocated in scoped memory areas are not garbage collected but instead the whole memory area is reclaimed when the program exits the scope in which the memory area was allocated. The access restrictions associated with scoped memory (e.g., objects allocated on the heap may not reference objects in scoped memory, and real time threads aren't allowed to access the heap[2]) make inter-thread communication more difficult. Real-time threads, however, may share scoped memory areas.

In RTCE, real-time objects are allocated in *core memory*, and may not access objects on the garbage collected *baseline* heap. Objects on the heap may, with some restrictions, access core objects through special method calls. Core objects are allocated in an *allocation context*. When an allocation context is released, all objects in it may be eligible for reclamation but, since there might be references from the baseline heap, the actual reclamation is done by the baseline garbage collector when all of the objects in the allocation context are unreachable. Thus, a non-real-time garbage collector is used to reclaim the memory used by the real-time processes.

In RTCE, there are no limitations on which allocation contexts objects may reference so it is up to the programmer not to release an allocation context when it is still referenced.

RTCE also specifies stack allocation of real-time objects, which are to be automatically reclaimed as the scope is exited. To allocate stack objects, a set of restrictions apply and the reference must explicitly be declared stackable.

Under both of these specifications, behaviour similar to our non-critical allocations can be achieved by using one memory area (or allocation context) for critical memory and another (or the heap) for the non-critical objects. The drawbacks of these approaches compared to the one proposed in this thesis are firstly that a much higher responsibility is placed on the programmer by removing the safety that garbage collection provides, from the most critical parts of the system. Secondly, the access restrictions between the different types of memory make communications between low and high priority threads more complicated.

[2]Since the heap is garbage collected, real-time threads with hard time constraints must be of the type *NoHeapRealTimeThread* in order to avoid interference from the garbage collector.

8.4 Soft references

The notion of non-critical allocation is somewhat related to the *soft references* found for instance in Java [26] in that they both aim to prevent out of memory errors due to too many objects not absolutely needed for the correct operation of the program. In analogy with the Java terminology, non-critical allocations could be called "soft allocations".

The difference lies in *when* the system decides that it is running low on memory and starts trying to limit memory usage. With the approach presented here, the decision is taken at allocation time, preventing a low on memory situation from arising. When using soft references, on the other hand, all allocations are carried out, and the decision about when to reclaim softly reachable objects are left to the garbage collector. There is also a difference in the intended usage; soft references were introduced to facilitate the implementation of e.g. caches, where objects' lifetimes are nondeterministic (i.e., you never know whether a cached value will be accessed again in the future or not, but it's best to keep it as long as the memory permits). Thus, while soft references may be used to achieve a similar logical behaviour as our non-critical allocations, the increase in the amount of required GC work when the system is already low on memory makes this use of soft references unsuitable for for real-time applications.

8.5 Worst case and schedulability analysis

Good worst case estimates for execution time and memory usage are crucial for making any kind of real-time guarantees. In order to make such analysis feasible in industry, tool support is required

Alan C. Shaw has developed a technique, *timing schema* [47], for formalizing execution time analysis. A timing tool for a subset of C has also been developed [38].

In order to give continuous feedback to the developer, an interactive programming environment with worst case analysis functionality is desirable. The experimental tool Skånerost [41] developed at our department provides interactive worst case execution time [40] and memory [39] consumption analysis based on timing schema and source code annotations for (currently a subset of) the Java language.

The WCET group at Uppsala University has presented research on and tool support for worst case analysis on C code without the requirement for programmer annotations based on flow analysis and pipeline simulation [18, 17].

Another approach to schedulability analysis and automatic verification of real-time systems based on timed automata has been developed in the UPPAAL project [33, 42].

Current approaches to worst case analysis are often highly complex when applied to life-size programs. A different approach to temporally predictable software is proposed by Puschner [43]. That approach is based on trading off performance for predictability by writing (or automatically transforming) programs in a way that they are inherently predictable; *single path programming*. It is not clear how this approach affects dynamic memory management.

CHAPTER 9

CONCLUSIONS

Motivated by the desire for greater flexibility in real-time systems, and the need to handle non-determinism and variations in resource utilization, new approaches to memory management have been presented.

A model for scheduling garbage collection work, *time-triggered GC scheduling*, that has several benefits compared to previous techniques is proposed. The single scheduling requirement that the garbage collector must finish before its deadline makes it especially suitable for earliest deadline first (EDF) systems, for which we have not seen any similar systems.

The handling of non-determinism and the desire to enable the run-time system to provide real-time performance without requiring worst case analysis motivated two approaches to adaptive memory management. Firstly, techniques to accomplish auto-tuning of a concurrent, real-time, time-triggered garbage collector were examined. Adaptive GC scheduling contains two orthogonal problems: to determine the scheduling parameters of the GC process and to keep a task set with varying resource utilization schedulable. Much work has been done in the feedback scheduling community on the latter problem so this thesis has focused on the former.

Another approach to handling non-determinism and increase robustness — applying priorities to memory allocation — was presented. The motivation for this is that if the computer is running low on memory, we want run-time system support for selecting the most important memory allocations, just as the process scheduler makes sure that the most important processes get precedence over less important ones if CPU time is scarce.

9.1 Contributions

Time-triggered garbage collection scheduling

A number of problems related to GC scheduling was addressed:

- Using time rather than allocation as the trigger for GC work solves the problem of bursty allocations causing long GC pauses. It also allows us to spread the GC work evenly across the GC cycle. In essence, by turning GC work into a periodic activity rather than a sporadic one, the scheduling of GC is simplified.
- The metric used to measure GC work has a big impact on the GC scheduling. The optimal GC work metric is the CPU time required to perform the GC work and it is proposed that it is practically possible to use time as the GC work metric at run-time.
- Implementing non-intrusive concurrent GC with guaranteed progress in an EDF scheduled system has been problematic. Time-triggered GC scheduling provides an explicit deadline for each GC cycle and therefore fits nicely into an EDF system.
- Time-triggered GC makes it possible to schedule the GC thread as any other thread. The GC work metric is only used for schedulability analysis and therefore, the problems of poor real-time performance caused by a poor metric are avoided.

These ideas form a novel approach to non-intrusive, concurrent garbage collection scheduling in real-time systems.

Adaptive garbage collection scheduling

As the time-triggered approach to garbage collection scheduling allows us to make scheduling decisions at the GC cycle level rather than individual increments, it lends itself well to auto-tuning. Techniques for estimating both the GC cycle time and the amount of GC work required to complete a cycle was presented and their applicability was experimentally verified.

Using time as the GC work metric facilitates the integration of the GC scheduler with a general feedback scheduler as time can be measured directly. Since the same parameters are used to schedule both GC and application threads the memory management overhead can be taken into account when performing on-line schedulability analysis in a straight-forward manner.

The proposed techniques can facilitate the implementation of more flexible real-time systems as they make is possible to use GC in a real-time system without the need for tedious manual tuning.

Priorities for memory allocations

Based on the observation that not all of the code in a hard real-time system is critical, the idea of applying priorities to memory allocation was presented. This can be used to enhance the robustness of real-time and embedded systems in two ways:

- It provides run-time support for prioritizing memory allocations if there is not enough memory for all allocation requests and thereby facilitates development of robust applications.
- It makes it easier to provide hard guarantees since the worst case memory usage only has to be analyzed for the critical parts of the system as non-critical allocations cannot cause the system to fail.

Furthermore, experiments also show that the same mechanisms can be used to increase performance by limiting the amount of memory allocation and, consequentially, garbage collection work.

9.2 Reflections

In the introduction, it was stated that an important property of a memory manager to be used in a flexible real-time system is that it should provide real-time performance without the need for a priori analysis. That is, if the total requested CPU utilization of mutator and collector is low enough that the system is schedulable, the actual schedule produced by the run-time system should allow all tasks to meet their deadlines. The inherent robustness of the time-triggered GC scheduling model and the property that low-level scheduling decisions are left to the process scheduler combined with the presented approaches to adaptive GC scheduling and memory allocation helps resolve the memory management issues of flexible real-time software.

The second goal was to develop a model that makes it possible to schedule garbage collection as any other thread while still guaranteeing sufficient progress. This thesis shows that time-triggered GC scheduling has this property under both fixed priority and EDF scheduling.

Garbage collection is essential to the use of safe object oriented languages in real-time systems and the contributions of this thesis are a step towards making real-time garbage collection practically feasible.

BIBLIOGRAPHY

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] Luca Abeni and Luigi Palpoli. On adaptive control techniques in real-time resource allocation. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 2000.

[3] Leif Andersson and Anders Blomdell. A real-time programming environment and a real-time kernel. In Lars Asplund, editor, *National Swedish Symposium on Real-Time Systems*, Technical Report No 30 1991-06-21. Dept. of Computer Systems, Uppsala University, Uppsala, Sweden, 1991.

[4] Aspect-oriented software development web site; http://www.aosd.net.

[5] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[6] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL'03*, New Orleans, Louisiana, USA, January 2003.

[7] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[8] Mats Bengtsson. *Real-Time Compacting Garbage Collection Algorithms.* Lic. eng. thesis, Department of Computer Science, Lund University, 1990.

[9] D. G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 11(3), July 1968.

[10] Greg Bollella et al. *The Real-Time Specification for Java.* Addison-Wesley, 2001.

[11] Giorgio Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3), March 2002.

[12] Anton Cervin. *Integrated Control and Real-Time Scheduling.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, April 2003.

[13] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1), July 2002.

[14] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12), December 1960.

[15] Ole Johan Dahl and Kristen Nygaard. *SIMULA – A language for Programming and Description of Discrete Event Systems.* Norwegian Computing Center, Oslo, Norway, 5th edition, September 1976.

[16] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11), November 1978.

[17] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Department of Information Technology, Uppsala University, 2002.

[18] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A worst-case execution-time analysis tool prototype for embedded real-time systems. In *Proceedings of the Workshop on Real-Time Tools (RT-TOOLS 2001)*, August 2001.

[19] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In *OOPSLA '91 GC Workshop*, 1991.

[20] R. Fenichel and J. Yochelson. A lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11), November 1969.

[21] Roger Henriksson. Adaptive scheduling of incremental copying garbage collection for interactive applications. In *Proceedings of the 1996 Nordic Workshop on Programming Environment Research (NWPER'96)*, Aalborg, Denmark, 1996.

[22] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[23] Anders Ive. *Implementation of an Embedded Real-Time Java Virtual Machine Prototype*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2003. (in preparation).

[24] Anders Ive, Anders Blomdell, Torbjörn Ekman, Roger Henriksson, Anders Nilsson, Klas Nilsson, and Sven Gestegård Robertz. Garbage collector interface. In *Proceedings of NWPER'02*, Copenhagen, Denmark, August 2002.

[25] J-Consortium. Real-time core extensions for the java platform. International J Consortium Specification, 2000.

[26] Java 2 platform, standard edition, API specification. Sun Microsystems. http://java.sun.com.

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

[28] M Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5), 1986.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the Europeean Conference on Object Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[30] Donald E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*. Addison-Wesley, 1973.

[31] Hermann Kopetz. Time-triggered real-time computing. *IFAC World Congress, Barcelona, July 2002, IFAC Press*, July 2002.

[32] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[33] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[34] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time enviro nment. *Journal of the ACM*, 20(1), 1973.

[35] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4), April 1960.

[36] M. L. Minsky. A lisp garbage collector algorithm using serial secondary storage. Memo 58 (rev.) Project Mac, M.I.T., Cambridge, Mass., December 1963.

[37] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. Real Java for real time – gain and pain. In *Proceedings of CASES-2002*, pages 304–311. ACM Press, October 2002.

[38] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5), May 1991.

[39] Patrik Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999.

[40] Patrik Persson and Görel Hedin. Interactive execution time predictions using reference attributed grammars. In *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, Amsterdam, The Netherlands, March 1999.

[41] Patrik Persson and Görel Hedin. An interactive environment for real-time software development. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000.

[42] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice.* PhD thesis, Uppsala University, 1999.

[43] Peter Puschner. Is worst-case execution-time analysis a non-problem? — Towards new software and hardware architectures. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*, Vienna, Austria, June 2002.

[44] Sven Gestegård Robertz. Applying priorities to memory allocation. In *Proceedings of the 2002 International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, June 2002. ACM Press.

[45] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In *Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems – 2003 (LCTES'03)*, San Diego, California, USA, June 2003. To appear.

[46] Lui Sha, Ragunathan Rajkumar, and John. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.

[47] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7), 1989.

[48] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 2002.

[49] G. R. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9), September 1975.

[50] P. L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9), September 1976.

[51] Paul R. Wilson, Mark S. Johnstone, Michal Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proc. 1995 International Workshop on Memory Management*, Kinross, Scotland, September 1995.

[52] Qian Yang, Witawas Srisa-an, Therapon Skotiniotis, and J. Morris Chang. Java virtual machine timing probes – a study of object life span and GC. In *Proceedings of 21th IEEE International Performance, Computing and Communications Conference (IPCCC)*, Phoenix, Arizona, April 2002.