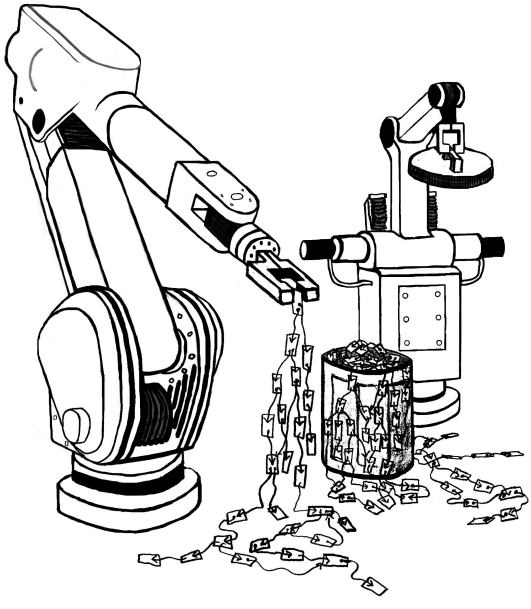
Automatic memory management
for flexible real-time systems

Automatic memory management for flexible real-time systems



Sven Gestegård Robertz

Doctoral dissertation, 2006

Department of Computer Science
Lund University

ISBN 91–628–6829–2
ISSN 1404 –1219
Dissertation 24, 2006
LU-CS-DISS:2006-01

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

Typeset using LaTeX 2ε

Cover artwork by the author, inspired by Wadler [Wad76]

Printed in Sweden by Tryckeriet i E-huset, Lund, 2006

Abstract

In a flexible real-time system, the constraints in available CPU time and memory lead to resource management problems, which must be handled carefully in order to maximize quality of service while avoiding overload. Managing CPU time — scheduling — is well studied and dynamic scheduling is widely accepted in the real-time industry. In order to make safe high-level languages, like Java, practically feasible for use in hard real-time systems, memory management and particularly the dependencies between memory and CPU usage must be studied.

The traditional approach to incremental GC scheduling, to perform garbage collection work in proportion to the amount of allocated memory, has drawbacks such as inconsistent utilization due to bursty allocations. To remedy this, *time-triggered GC scheduling* is proposed. It is shown that this strategy gives 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 complex and error-prone work metrics.

Time-triggered GC also allows a more high-level view on GC scheduling, as the entire GC cycle is considered rather than each individual increment. This makes it possible to schedule GC as a normal task. As the scheduling parameters are explicit in the model, it also makes the time-triggered strategy well suited for auto-tuning and fits well into feedback scheduling systems.

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.

Together, these solutions facilitate flexible and robust automatic memory management for real-time systems. Adaptive techniques are presented, aimed at replacing or complementing *a priori* analysis with online auto-tuning. The presented ideas have been successfully implemented and validated in an experimental real-time Java environment, supporting the claim that this work is a step towards *write once — run anywhere* with hard real-time performance.

Acknowledgments

I wish to thank my supervisors, Boris Magnusson and Klas Nilsson for their support and for giving me the freedom to choose and pursue research ideas and directions I have found interesting. I am also most grateful to my assistant supervisor Roger Henriksson, who introduced me to the field of real-time garbage collection. Görel Hedin, my supervisor in a previous project, at the beginning of my graduate studies, taught me much about research and technical writing. Thank you all for your patient help, valuable input, and encouragement throughout this work.

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. Thanks to Anders Ive for his help with implementing some of these ideas in the IVM virtual machine, Anders Nilsson for help with the LJRT compiler, Mathias Haage for the virtual robots, and Anders Blomdell for expert help with the embedded PowerPC platforms and insightful comments on low-level run-time system implementation. The LJRT compiler was developed using the JastAdd compiler tools by Görel Hedin, Torbjörn Ekman, and Eva Magnusson.

I also wish to thank Anton Cervin and Dan Henriksson for valuable discussions on scheduling and control systems, Torbjörn Ekman and Patrik Persson for interesting and enjoyable discussions on programming languages and real-time systems development, Ulf Asklund for teaching me a great deal about configuration management, and Christian Andersson for assisting with LaTeX tips and tricks.

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

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.

The presented research has been a collaboration between the Department of Automatic Control and the Department of Computer Science at Lund University, and was carried out within the research project "Integrated Control and Scheduling" for which Karl-Erik Årzén, Klas Nilsson, and Ola Dahl wrote the original proposal, and the "FLEXCON — Flexible Embedded Control Systems" research program. 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).

Finally, I would like to thank my friends, the members of the academic symphony orchestra, and all crazy people in spex and spääx for making my years as a student in Lund thoroughly enjoyable, my parents for always being there for me, and Kerstin for her love and support.

Lund, April 2006

Sven

CONTENTS

CHAPTER 1

INTRODUCTION

Today, computers are used as components in all kinds of systems and products, from industrial robots and cars to home appliances and toys, and functionality that has traditionally been implemented using mechanical systems or analog electronics now often include a computer or even a network of computers. Such *embedded systems* typically need to interact with an external evironment, and the dynamics of the environment give rise to timing constraints on program execution. Therefore, most embedded systems are also *real-time systems*, meaning that they have to react to external stimuli within a specified time.

In a system with timing requirements, all parts of the system — including the run-time system — must be implemented in a way that makes them temporally predictable. The overall goal of this thesis is to develop new techniques to improve the real-time properties of run-time systems for embedded and real-time applications, particularly with respect to memory management,

As the complexity of embedded software increases, so does the engineering and programming effort required. High-level programming languages provide programmer-friendly abstractions that hide much of the low-level details, notably memory management. Apart from making programming easier, that also improves safety and robustness, as some types of common programming errors are simply not possible to make at the abstracted level. The drawback is that, at the higher level of abstraction, the programmer no longer has full control of all low-level aspects of the system. When responsibility for handling low-level tasks is transferred from the programmer to the run-time system, predictability must not be lost, and engineering decisions traditionally expressed directly in code must be possible through e.g. parameters to the run-time system.

1.1 Resource-aware computing

A fundamental property which makes embedded software different from computer programs in general is that an embedded application must, to a much higher degree, execute in a resource-constrained environment. Therefore, we will very briefly examine the issues associated with resource-constrained applications, in order to help putting the presented work into context.

In general, resource constraints on computer systems can be considered to fall into five cathegories, of which the first two (and, in particular, the dependencies between them) and the last one are of primary interest in this thesis;

1. CPU time
2. Memory
3. Input/Output capabilities (I/O, networking, etc.)
4. External physical (Energy, power, space)
5. Engineering effort

The first four are physical constraints that obviously cannot be violated and therefore must be taken into account in the development of an application. The engineering effort required, on the other hand, addresses the economic aspects of the development; just as there are trade-offs between the physical constraints (e.g., using a faster CPU or larger memory may allow a more sophisticated algorithm to be used, but it comes at the cost of higher power consumption), programming in a high-level language may result in a bit less efficient code than hand-written assembly language, but the development time, and the number of errors, would most likely be much lower. Therefore, economical or time-to-market concerns may favour a high-level language even if it incurs e.g. additional run-time overhead and higher hardware costs.

Resource constraints are extra-functional requirements, and should ideally be handled independently of the functional requirements. That is further emphasized by the increasing desire to use modular, or component based, methods of development; when the individual modules or components are developed, it is not known how they will be composed into the final system, and therefore their implementation must not depend on such knowledge.

Instead, resources should be managed by a *resource manager*, a component of the run-time system. In the real-time community, the resource

management problem applied to CPU time has been thoroughly studied and the theoretical foundation of on-line scheduling is well built. Recently, with the development of processors with variable clock frequency, the relation between CPU time and power consumption has been investigated. Memory management, on the other hand, has been regarded as part of the application, and has not been considered in this context. While viable in traditional systems, the introduction of automatic memory management complicates the picture as illustrated by the following sketch of past, present, and future software architectures.

Traditionally, a real-time system uses on-line process scheduling and static or manual memory management. Figure 1.1 illustrates how this architecture provides temporal isolation both between different applications and between applications and the run-time system; the only resource that is managed is CPU time, and the process scheduler has full control over the CPU time assignment. Thus, one application overrunning its designed execution time cannot cause a failure of another that is executing with higher priority on the same CPU.

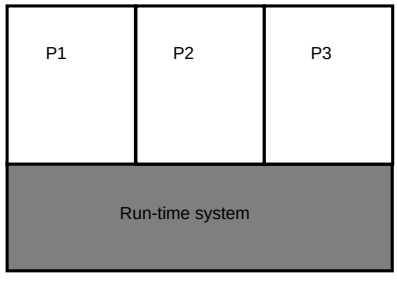


Figure 1.1: *Traditional model, whith independent processes running on top of a run-time system.*

Introducing automatic memory management, as a part of a safe language, complicates the picture, and the boundary between the application and the run-time system becomes unclear as shown in Figure 1.2. There are multiple reasons for this; First, memory is a global resource, and without global management, over-use of memory in one part of the system may cause failure of another part [1]. Secondly, most current real-time garbage collectors (RTGC) are intrusive, in the sense that they can interrupt the application threads at arbitrary times, causing latencies and jitter. Scheduling of garbage collection work also often by-passes

[1] While the focus of this thesis is on memory management, similar problems arise from shared use of any shared global resource for which the system does not provide arbitration.

the normal task scheduler, further compilicating things. Finally, the CPU requirements of the GC depends heavily on the behaviour of the application, which complicates off-line schedulability analysis, as worst case analysis would require global analyis of both memory and CPU usage of both the applications and the GC.

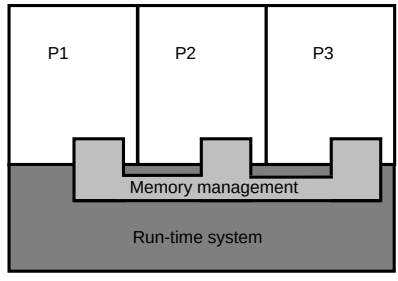


Figure 1.2: *Introducing automatic memory management into a real-time system complicates the boundary between applications and run-time system*

The problem of resource management is that the utilization of different resources cannot be handled independently, as performing a certain task typically requires simultaneous use of several resources. In order to get the clear separation of the traditional model, future run-time systems will need to include some sort of resource manager, as in Figure 1.3. With control over resource utilization, the dependencies between different resources can be taken into account, resulting in the desired isolation between applications and the run-time system. That is an emerging trend, and research on quality-of-service and quality-of-control has resulted in techniques and mechanisms for such isolation of applications with respect to CPU time and I/O capabilities. Investigating how memory management concerns can be incorporated into that picture is part of the motivation for the presented work .

1.2 Memory management

Memory management in real-time and embedded systems is still handled in a very conservative manner and for reasons of safety and predictability, static memory management is often the technology of choice. However, as the complexity of embedded systems increase, static implementations become problematic; they are difficult to maintain and develop as even minor changes may require major reorganization of the software, and resource utilization may be low. For those reasons,

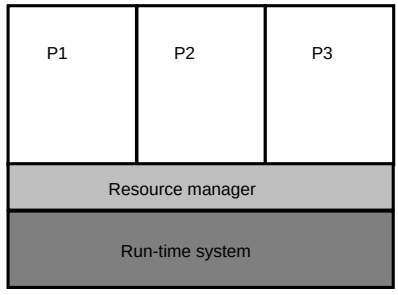


Figure 1.3: *A resource manager provides the desired isolation between applications and run-time system*

dynamic memory management becomes increasingly desirable. While more flexible than static memory management, manually managed dynamic memory introduces 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 GC scheduling aimed at meeting these demands.

The focus of this thesis is on GC scheduling rather than algorithm design, and the fundamental idea is to let elapsed time, rather than performed allocations, determine when to run the garbage collector, using an approach called *time-triggered garbage collection*. 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.

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.

1.3 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. Two categories of problems are addressed: 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. 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 the real-time behaviour of a system. 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[2]. Therefore, running out of memory is likely to cause the entire system to fail while requesting too high CPU utilization may cause some or all processes to miss deadlines but the system may be able to continue executing with decreased performance.

[2]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. Exceptions of course exist, for instance large embedded systems like ships and power plants. However, in such systems, virtual memory should not be used for the time-critical tasks as it reduces predictability. One method of ensuring this is to lock the pages used by hard real-time tasks into RAM in order to avoid page faults.

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 systems that have a safe failure mode, a small theoretical risk of failure 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 that it will execute with 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.

The first issue 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.

1.4 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 results 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 on-line estimation of 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: Memory-aware feedback scheduling presents an approach to extending traditional feedback scheduling results to also incorporate the costs of memory management explicitly in the period time optimization.

Chapter 7: GC in an uncooperative environment describes the challengenges faced when implementing accurate concurrent GC in an environment where one cannot rely on cooperation from the compiler back end or scheduler.

Chapter 8: Experiments presents experimental support for the proposed techniques.

Chapter 9: Future work outlines directions for future research and points out possible areas of application for the presented ideas.

Chapter 10: 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 11: Conclusions summarizes and discusses the contributions of this thesis.

Publications

This thesis is largely based on papers published, or under submission. The ideas of time-triggered GC and on-line estimation of GC scheduling parameters of Chapter 3 and Chapter 4 include results from

> Sven Gestegård Robertz and Roger Henriksson, *Time-Triggered Garbage Collection — Robust and Adaptive Real-Time GC Scheduling for Embedded Systems* [RH03], in Proceedings of the ACM SIGPLAN Langauges, Compilers, and Tools for Embedded Systems – 2003 (LCTES'03).

Chapter 5 and the corresponding experiments was published as

> Sven Gestegård Robertz, *Applying Priorities to Memory Allocation* [Rob02] in Proceedings of the 2002 International Symposium on Memory Management (ISMM'02).

Chapter 6 is based on

> Sven Gestegård Robertz, Dan Henriksson, and Anton Cervin *Memory-aware feedback scheduling*, to be submitted.

Chapter 7 includes results presented in

> Anders Nilsson and Sven Gestegård Robertz, *On real-time performance of ahead-of-time compiled Java* [NR05], in Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'05).

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* [IBE+02], in Proceedings of NWPER'02.

There is also a close relation between the development of control application prototypes and and the Lund Java-Based Real-Time (LJRT) platform which is described in

> Anders Nilsson and Sven Gestegård Robertz, *LJRT compiler reference manual*, Department of Computer Science, Lund University, 2005 – 2006.

The LJRT platform is also central to the ideas presented in

> Sven Gestegård Robertz, Anders Nilsson, Klas Nilsson and Mathias Haage, *Multi-stage deployment of robot control software* [RNNH06], to appear in Proceedings of the 8th International IFAC Symposium on Robot Control, SYROCO, September 2006.

CHAPTER 2

PRELIMINARIES

This chapter briefly presents the fundamental concepts of real-time and embedded 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

In order to understand the problems and challenges associated with memory management in real-time systems, we will now review the fundamental properties of systems with timing requirements. We will discuss what defines a real-time system, how timing requirements arise and are classified, how a set of processes may share a single CPU while still performing in a timely manner and how processes with firm timing requirements can co-exist with non real-time processes.

For any computer program, its task is, generally speaking, to produce some output based on its input values. The fundamental definition of correctness is that the program produces the right output for any valid input values. However, for many systems, typically those that interact with an external environment in some way, that 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 Concurrent programming

Concurrent programming is the common name for the techniques used to allow many processes to execute in parallel on the same computer, either truly in parallel on a multi-CPU machine, or virtually so by time-sharing on a single CPU, in a consistent way. Important problems are how to deal with communication and synchronization between parallel activities. There are several reasons for using concurrent programming, but for our purposes, the most important one is to model parallelism in the external environment. A program in a control system may need to react both to occurrences of certain events and perform operations at certain times. If the different events are independent of each other and of the passage of time, there are parallel activities in the controlled system, and therefore it is convenient to have the same parallelism in the software. We will not go into any details on the different problems in concurrent programming or their solutions. For now it will suffice to state that concurrent and real-time programming are very tightly related (and sometimes even used synonymously); virtually any real-time or embedded system will consist of many processes, either cooperating or independent.

If there are more parallel processes than there are processors, like when running multiple processes on a single-CPU machine, not all processes can execute simultaneously with true parallelism. Therefore, some form of time-sharing mechanism must be used to allow them to seemingly execute in parallel, by switching back and forth between processes, interleaving their execution. Time-sharing can be implemented either explicitly in the processes, e.g. using *co-routines* [Knu73], or by the run-time system or OS. In the latter case, a special piece of software called the *process scheduler* is responsible for selecting which process that gets to execute next.

Normally, the scheduler runs periodically, and at each invocation it suspends, or *preempts*, the running process and selects another process, which is then allowed to execute until the next scheduler period. This model is called *time-slicing*. How processes are scheduled obviously affects their temporal behaviour, and different scheduling techniques are presented in Section 2.1.4.

2.1.2 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 for the interaction to work well. If the system takes seconds or more to respond to 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 react 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.

Specifying temporal behaviour

Having defined a real-time system as a system that must react in a timely manner, we will now discuss how timeliness can be parametrized and measured. The real-time behaviour of a process can be specified as a tuple $(\mathbb{R}, \mathbb{C}, \mathbb{D})$, where $\mathbb{R}$ is the set of *release* times, $\mathbb{C}$ is the set of *execution* times, and $\mathbb{D}$ the set of *deadlines*. For a periodic process with start time t_0 and period time T, a constant execution time C and deadline D, we get

$$\begin{aligned} \mathbb{R} &= \{R_k\} = \{t_0 + kT\} \ ; \ k \geq 0 \\ \mathbb{C} &= C \\ \mathbb{D} &= \{D_k\} = \{t_0 + kT + D\} \ ; \ k \geq 0 \end{aligned} \tag{2.1}$$

which is the form we will assume if nothing else is stated.

In addition to the fundamental real-time requirement — that a process always finishes before its deadline — there are other aspects that are of interest when specifying real-time systems, namely *latency*, *response time*, and *jitter*. When a process is released, it may not always start executing directly; for instance, if another process is executing it will have to wait until that process has finished. Therefore, the actual *invocation* time will be some time after the release, and the difference is called *latency*. The *response time* of a process is defined as the time from release to finish. These definitions are illustrated in Figure 2.1.

Finally, the variations in these quantities from one period to another, or *jitter*, may be important. Figure 2.2 shows an example of jitter in both latency and response time. Process 1 is periodic with period time T, and has the release times $\{t_0 + nT\}, n \geq 0$. However, in the example, the process actually starts executing in t_0, $t_0 + T + L_2$ and $t_0 + 2T + L_3$, and the execution is not exactly periodic. If we define latency jitter as the difference between the minimum and maximum latency, in this case we

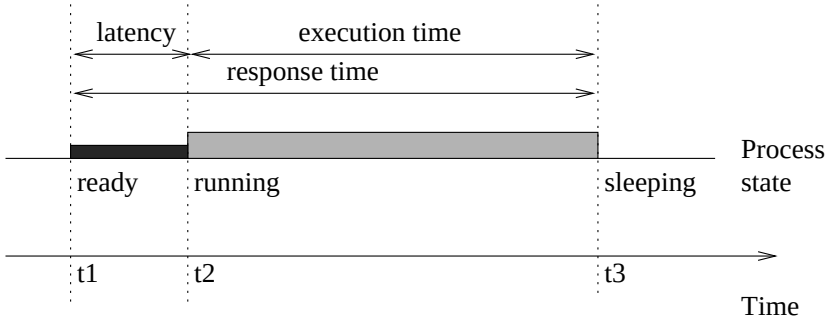


Figure 2.1: *Definitions of real-time parameters. t_1 is the* release time *of the process, at t_2 the process is* invoked*, and at t_3, the process has finished its execution and sleeps until its next release. The time from release to invocation is called* latency*, and the time from release to finish is called* response time*.*

get the maximum jitter $\Delta L_{max} = L_2 - 0 = L_2$. In the example, there is also jitter in the response time, defined analogously.

It should be stressed that real-time systems are a very heterogenous class of systems, and there are large variation in which aspects are important. For instance, one application may be very sensitive to jitter whereas in another, only the response time is important. Also, there are vast differences in time-scale. A typical video application has a sampling rate (frame rate) in the range of 25 – $100\,Hz$, while control applications may have sampling rates of tens of kilohertz. Therefore, the fields of real-time systems research and engineering is also quite heterogenous, as the different requirements will give rise to different technical solutions.

2.1.3 Control systems

A large class of embedded systems are *control systems* [AW97], where the task of the computer is to control the behaviour of some external, physical, process. When implementing a controller for a continuous-time process on a computer, the process must be sampled, and most of the control theory assumes that the samples are taken at a constant rate, i.e. *periodic sampling*. Therefore, it is common to implement controllers as periodic processes.

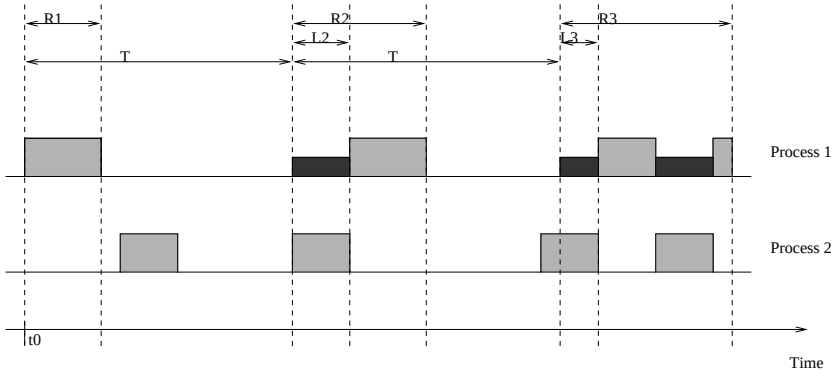


Figure 2.2: *Example of latency and response time jitter. Process 1 is periodic with time T, Process 2 is sporadic and has higher priority. In this figure, R denotes response time and not release time.*

In controller design, it is common to discretize the continuous-time system under the assumptions of periodic sampling and constant input-output delay. Any jitter, in the sampling or output times, will cause errors in the linearized model, which in turn leads to degraded performance of the controller. Note, however, that it is not the jitter or delays in the computer task, but in the sampling and control action, that has impact on control performance.

In analog with the previously defined real-time parameters, for control systems we add corresponding quantities directly related to the control task. Figure 2.3 shows the execution of one invocation of a controller. In addition to the latency and response time, the control counterparts are defined. Figure 2.4 illustrates how the interactions between processes affect both latency and response time. Controller 2 is executing when Controller 1 is released which causes latency to Controller 1. Controller 2 has higher priority than Controller 1 and is therefore allowed to *preempt* Controller 1, which increases the response time of Controller 1.

This illustrates how timing requirements on the controller process come from the assumptions made in the controller design. For instance, if a small sample delay is desired, the latency of the control task must be small. The integration of control design and real-time scheduling has been studied [Cer03], tools for analysis of the effects of varying delays have been developed [LC02] as well as theoretical results for taking jitter into account when doing control design [CLE+04].

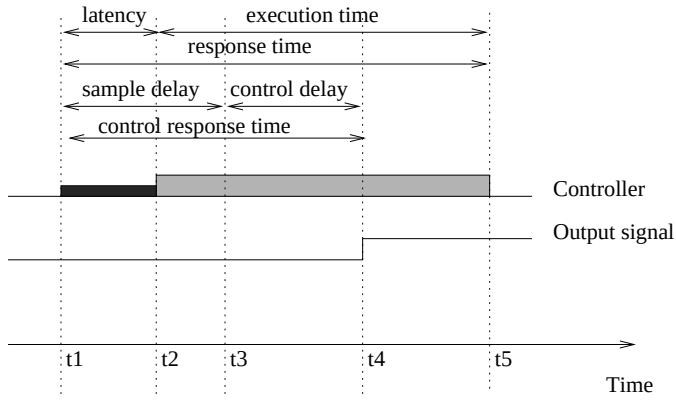


Figure 2.3: *Definitions of timing parameters for the control task. At t_1 the controller is released, at t_2 it is invoked, at t_3, the input signal is sampled, at t_4 the new control signal is output and at t_5 the process has finished its execution.*

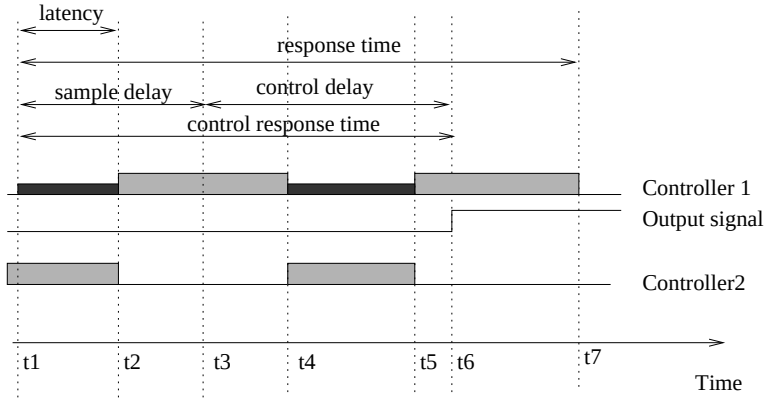


Figure 2.4: *Interference from other processes affect real-time behaviour. Controller 1 is released at t_1, but does not get to execute until t_2, when Controller 2 has finished. Then, at t_4, Controller 1 is preempted by Controller 2 and is suspended until t_5.*

2.1.4 Predictability and scheduling

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.

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*[1] *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 scheduling algorithms will be given.

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 suspended 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.

A system is schedulable if all processes are guaranteed to meet their deadlines, i.e. that their worst case response time is less than the deadline. The fundamental result in fixed-priority scheduling is that if all processes are released at the same time (known as a *critical instant*), the system is schedulable if all processes finish before their deadline. RMS is optimal in the sense that if a set of processes are not schedulable with priorities assigned according to RMS, it will not be schedulable for any

[1] A note on terminology: Here, *static* and *dynamic* are used with respect to the *schedule* itself. Other terms are off-line and on-line scheduling. This should not be confused with the taxonomy used by Liu and Layland [LL73]. They discuss the distinction between static and dynamic scheduling algorithms based on whether *priorities* are fixed or may change during execution. In that context RMS and DMS are static algorithms, wheras EDF is dynamic.

other set of priorities. It can be proved that for n independent, periodical processes, with execution time C_i and period time T_i, a RMS system is guaranteed to be schedulable if

$$\sum_{i=0}^{n} \left(\frac{C_i}{T_i} \right) < n \left(2^{1/n} - 1 \right) \tag{2.2}$$

From this, it follows that, for an arbitrary number of processes, a RMS system is schedulable if the total CPU utilization is less than 69% [LL73]. The assumptions in this result are quite restrictive and not directly applicable for exact analysis in practice. It is still, however, a good rule-of-thumb to be used as a starting point.

In the rate monotonic case, the deadline is assumed to be equal to the period time. For tasks where the response time is important, it is common to have a deadline that is shorter than the period time. In that case, priorities may be assigned based on their deadlines rather than their period times — *deadline monotonic scheduling*. If $D = T$, DMS and RMS are obviously equivalent. If $D < T$ it has been proved that DMS is optimal, in the above sense.

For detailed analysis of real systems, the assumptions must be relaxed. In particular, processes are seldom unrelated; either they cooperate in order to perform a common task, or they compete for some common resources. In both cases, they may interfere with each other in a way that breaks the assumptions behind (2.2). This is dealt with in the *generalized RMS* [SRL94]. The schedulability criterion is the same, all processes should have a worst case response time shorter than their deadline, but the response time calculations are extended to take blocking, while waiting for shared resources, etc., into account.

Earliest deadline first scheduling

Another approach to dynamic scheduling is *deadline driven scheduling* [LL73], also known as *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 — the translation from timing requirements to priorities is done 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.

It should be noted that, as there are no strict priorities in an EDF system, it may cause more jitter to high frequency tasks, compared to RMS; the fast tasks may occasionally be preempted by slower tasks that are closer to their deadline. This, however, only occurs in a heavily loaded system.

2.1.5 Co-existence of hard and soft processes

An important property in real-time and safety-critical systems is *isolation* between processes. In management of global resources, it is desirable to have a model that ensures that an overrun or violation of design-time assumptions in one part of the system cannot cause a shortage of resources in other parts of the system. Applied to scheduling, that means that, in a system with independent processes, if one process overruns its allocated execution time, it should not be allowed to "steal" CPU time from other 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 [AB98]. 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 blocked until the next CBS period. 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 al-

ways meets their deadlines by isolating them from non-deterministic processes while still allowing 100% CPU utilization.

The control server model

The control server model [CE03] is an attempt to combine the predictability of static scheduling with the flexibility of dynamic scheduling aimed at ensuring jitter-free execution of control tasks. The basic idea is to use static scheduling for input and output while the computations in between are scheduled dynamically, with EDF. Isolation of tasks is provided through a mechanism similar to CBS.

The control server model was designed to facilitate component-based development of control systems. The characteristic property of the model is that all parameters effecting control performance (latency, response time, period time, etc.) are linear in CPU utilization. This enhances composability, as each component only has one knob, CPU utilization, that needs to be tuned when the system is assembled.

2.1.6 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, 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* [AP00, Cer03, CEBÅ02]. 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 2.5 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 set-point, 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* [BLA02], where a stiffness value is assigned to each process and the scaling of period times is done in proportion to that value.

The general period assignment problem can be expressed as follows. A set of n tasks, $T_i, i \in \{1 \dots n\}$ with execution time C_i, an adjustable

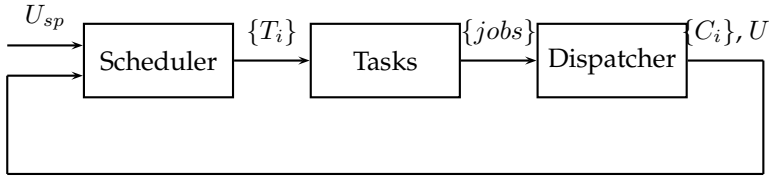


Figure 2.5: *The structure of a basic feedback scheduler. The scheduler measures the execution time of each process, C_i, and the total CPU utilization, U. The period times of the tasks, T_i, are scaled to achieve the setpoint utilization, U_{sp}.*

period h_i, and a cost function $J_i(h)$ share the same computer. The task of the feedback scheduler is to assign new sampling intervals $h_1 \dots h_n$ so that the total cost is minimized and the total CPU utilization is kept below a set-point, U_{sp}. This is formulated as the optimization problem

$$\begin{aligned} &\min_{h_1 \dots h_n} \sum_{i=1}^{n} J_i(h_i) \\ &\text{subject to} \sum_{i=1}^{n} \frac{C_i}{h_i} \leq U_{sp} \end{aligned} \tag{2.3}$$

In that formulation, the cost only depends on the sampling rate. More elaborate models, where also the state of the plant is taken into account have been developed [HC05].

2.2 Embedded systems

An embedded system can be defined as *a system that has a computer but is not in itself a computer*, and this is currently the dominating use for computers, accounting for a vast majority of processor sales. Embedded systems are found all over the range from tiny to very large systems, and examples include intelligent price tags, home appliances, toys, mobile phones, industrial robots, cars, aircraft, ships, and power plants. Therefore, one must be careful when making general statements about embedded systems and their properties. Design and implementation of embedded system is also a vast research area, and most of it is outside the scope of this thesis. Nonetheless, we will now briefly examine some of the key differences between embedded systems and general-purpose computers. The discussion will primarily target the small to medium

sized range of systems, with single CPU computers[2] and memory in the range from several kilobytes to a few megabytes[3], that are commonly found in e.g., process and robot controllers.

There is a strong connection between the fields of real-time systems and embedded systems, as most real-time systems are embedded and vice versa. Therefore, when studying real-time systems, it is often necessary to also consider the special properties and requirements of embedded systems. In addition to timing requirements this includes safety aspects and resource constraints.

2.2.1 Safety and dependability

A program in an embedded computer typically runs "forever" and is not directly accessible to the user. This puts stronger demands on robustness and dependability on embedded software as compared to, e.g., desktop applications[4]. While it is annoying if a word processing application occasionally crashes, users often accept having to reboot their PC once in a while. This is not the case for embedded systems; it would be unacceptable if the software of a microwave oven crashed and required the user to pull the plug to turn off the appliance.

The fact that the program never terminates means that even minor errors may, in time, cause a fault. For instance, a small memory leak in a desktop application will probably not cause any problems, as that memory will be returned to the system when the application is terminated. In contrast, in a program which never terminates, like an embedded system or a server application, even a small memory leak will eventually cause the system to run out of memory and fail.

Viewing the problem from a slightly different perspective, *fault tolerance* is an important aspect of embedded systems design. I.e., the system should always be able to reach a safe state if a fault occurs. For some systems, this may simply mean to emergency stop in case of a fault, or to use a watchdog mechanism to automatically reboot a computer if it stops responding. However, for many systems this is not possible, as they do not have a simple safe state. For instance, in a moving car with a drive-by-wire system, simply turning off the power does not leave the

[2] In the cases where more than one CPU is required, we use separate computers communicating via shared memory or a real-time network.

[3] Our platforms for experiments include the Atmel ATmega128 microcontroller with an 8 bit, 8 Mhz, 8 MIPS RISC CPU and 32 kilobytes of RAM and PowerPC G3 boards with 300 MHz CPU and 32 megabytes of RAM.

[4] A desktop computer can, of course, be part of a large embedded system, but usually not of its time-critical parts.

system in a safe state — it must be actively stopped. When using embedded computers in such systems, care must be taken to ensure that the critical parts of the software always will be able to continue executing, albeit in a "safe mode" with degraded performance.

With increasing system complexity, especially in software, the engineering effort required to ensure fault tolerance increases rapidly. Therefore, run-time systems and development tools for embedded applications must provide as much support as possible for making software safe and robust.

2.2.2 Well-defined area of application

The additional extra-functional requirements on embedded software increase the complexity of system design and makes software design more demanding. On the other hand, in contrast to a general-purpose computer, an embedded system is specifically designed to perform a number of well-defined tasks. This means that all aspects of the embedded hardware and software may be tailored for a particular task, making some aspects of software engineering easier.

For instance, while desktop or server applications may be subjected to widely varying workloads, embedded software in time-critical applications commonly operate in steady state most of the time, with distinct mode changes. Therefore, some problems that are undecidable in the general case, like worst case execution time or live memory analysis, may be practically feasible in an embedded systems context as problematic program constructs like unbounded loops are typically avoided.

2.3 Memory management

Memory management consists of two principal tasks; memory *allocation*, which means mapping a variable or an object to a particular memory address, and *de-allocation* or *reclamation*, to return the chunk of memory occupied by a variable or an object to the system so that it can be used to satisfy another allocation request. Memory management can be static or dynamic, and the latter is further divided into manual and automatic memory management. We will now briefly review these different techniques and the fundamental concepts of 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. With the exception of some parts of safety critical applications, static memory management is seldom used due to the low flexibility and difficult development and maintenance, and low average resource utilization which often results from using off-line techniques.

Dynamic memory management [Knu73, WJNB95] 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* and both are caused by the difficulty of manually determining object lifetimes; 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 program failure.

Determining when an object should be deallocated in order to avoid both memory leaks and dangling pointers is non-trivial in a complex system. To ensure memory consistency, systems with manually managed memory require rigid coding conventions and protocols for when allocation, deallocation and pointer passing is allowed.

In systems with *automatic memory management*, the task of keeping track of when an object is no longer in use and can be safely deallocated is preformed by the run-time system, which frees the programmer from this complex and error-prone task. The technique used to identify and reclaim dead objects is called *garbage collection (GC)*. Examples of early programming languages with GC are LISP [McC60] and Simula [DN76].

2.3.1 Garbage collection

There are different approaches to implementing GC [JL96]. In this thesis, we will only consider *tracing* collectors — collectors that traverse the reference graph in order to determine which objects are live and which are not. Examples are mark-sweep [McC60] and copying [Min63, FY69] collectors. Another approach to garbage collection is *reference counting* [Col60], where the idea is to keep a count of how many references there are to each object and reclaim objects when the reference count reaches zero [5].

This thesis focuses on scheduling of GC work, rather than GC algorithm design or implementation, and the presented approach is applicable to any concurrent tracing garbage collector. However, different collectors have different properties and different requirements on the collector/mutator interface. Therefore, we will now briefly review the most common fundamental algorithms for tracing garbage collection.

GC algorithms can be divided into *moving* or *non-moving*. If a non-moving collector is used, objects reside at the same address from allocation to reclamation, just as in manually managed memory (e.g., `malloc` and `free`). A moving collector, on the other hand, may move objects during collection in order, for instance, to *compact* the heap by moving all live objects to one end of the heap, leaving a single contious area of free memory, and thus avoiding (external) fragmentation. *Mark-sweep* is an example of a non-moving algorithm, and *mark-compact* and *copying* algorithms are moving.

The cyclic nature of garbage collection

Most (tracing) garbage collectors need to make multiple passes in order to identify the live objects and reclaim the garbage. For example, a mark-

[5] A problem with reference counting is that it cannot reclaim cyclic structures; even if a set of objects are no longer reachable from a program, cycles in the object graph will prevent reference counts from reaching zero, thereby preventing unreachable objects from being reclaimed. For this reason, pure reference counting is not suitable for embedded or other long-running software, where even small memory leaks will eventually cause the system to fail. There are, however, reference counting real-time GCs, including techniques for reclaiming cyclic structures [Rit03]. Cyclic structures are reclaimed either manually (by manually breaking cycles or using weak references), or by having a tracing GC as back-up. The former approach just transfers the responsibility to the programmer, and the latter still requires a tracing real-time GC to ensure real-time performance. Also, in a reference counting GC, the scheduling of GC work is implicit in the algorithm. Reference counts are updated at reference assignments, and thus performed in-line with the mutator code. For these reasons, reference counting is outside the scope of this thesis.

sweep collector first scans all root pointers[6], then marks live objects and finally sweeps the heap. We call all the activities required to identify and reclaim garbage a *GC cycle*. E.g., in the mark-sweep case, a GC cycle consists of root scanning, pointer traversal and sweeping.

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 for use in scheduling analysis 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.

Mark-sweep

Mark-sweep is the classic non-moving tracing collector. A GC cycle consists of two phases: In the *mark* phase the object graph is traversed and each visited object is marked as live. Then, during the *sweep* phase, all objects on the heap are examined and those that have not been marked are reclaimed.

Figure 2.6 shows an example of a heap before and after a mark-sweep GC cycle. As it is a non-moving algorithm, the free space is non-contigous after the GC cycle. The free blocks are typically linked together to form a *free list*, just as in traditional memory allocators.

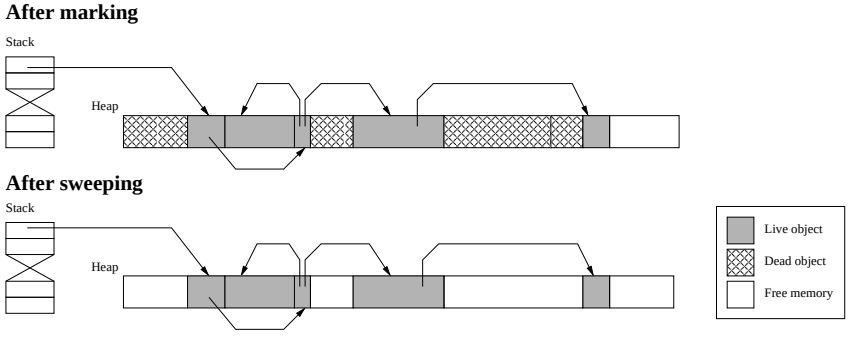


Figure 2.6: *Example of a heap before and after a mark-sweep GC cycle.*

[6]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.

Mark-compact

Mark-compact, is a moving version of mark sweep, where the heap is compacted in order to get one large contigous area of free memory, which both avoids external fragmentation and makes allocation simpler. The mark phase is the same as in mark-sweep, but instead of reclaiming the dead objects, the live ones are moved. Compaction can be done in different ways, and the one described in Figure 2.7 is *sliding* objects. In the compact phase, for each "hole" in the heap, the next live object is moved to the start of the hole, leaving one large chunk of free memory at one end of the heap. When objects have been moved, references are updated to point to the new location of the object.

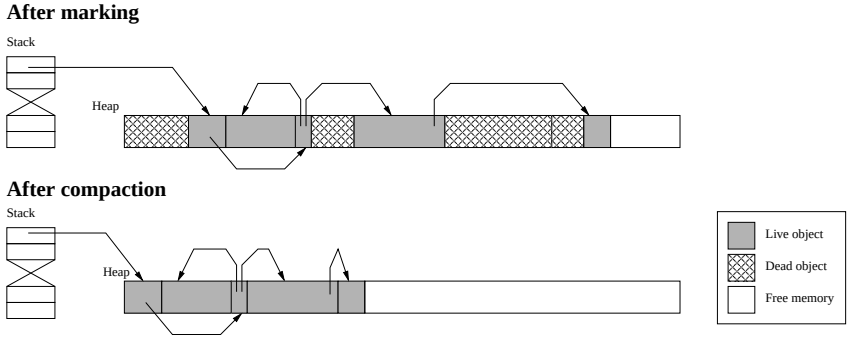


Figure 2.7: *Example of a heap before and after a mark-compact GC cycle.*

Copying collectors

Copying collectors, or *semi-space* collectors, in their basic form, work by dividing the heap into two halves. New objects are allocated in one space, *fromspace*, until it is filled up. Then, the reference graph is traversed and all live objects are *evacuated* into *tospace*. When an object is evacuated, a *forwarding pointer* in the fromspace copy points to the new location. When the heap is scanned, all encountered references pointing to the fromspace copy of an opbject are updated to point to the tospace copy. Finally, the spaces are *flipped*, so that the old tospace becomes fromspace, and vice versa. After the flip, the new tospace contains one big area of free memory. Figure 2.8 shows an example of a GC cycle with a copying collector.

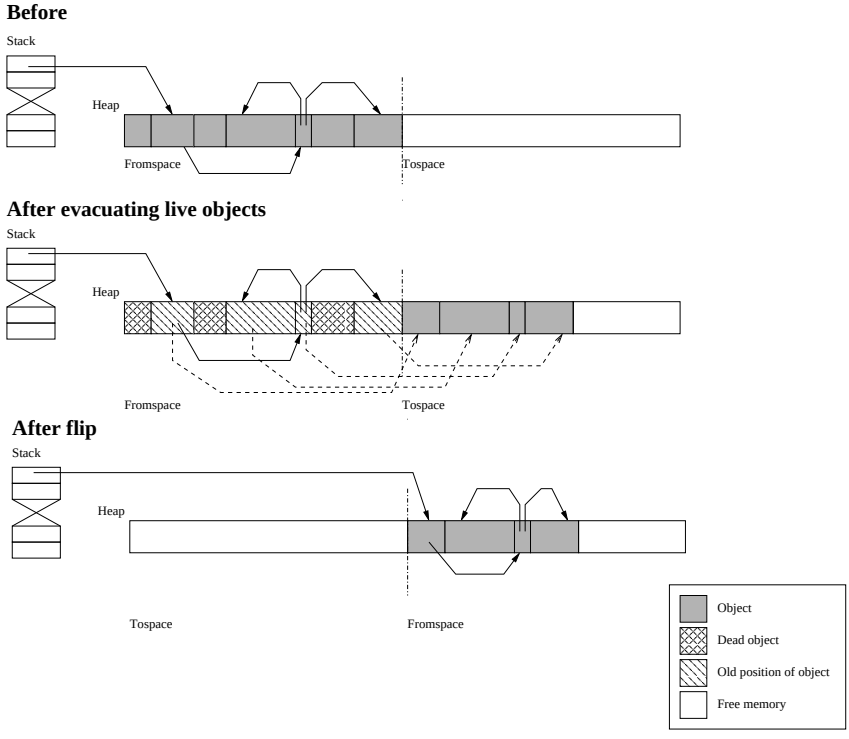


Figure 2.8: *Example of a GC cycle in a copying collector. First, the reference graph is traversed and the live objects evacuated to tospace. Then, the flip is performed, and the memory of (the old) fromspace is reclaimed. This is a schematic illustration of the principle of operation. In a real collector, references are updated as they are encountered during evacuation, and the relative positions of objects may differ between fromspace and tospace.*

2.3.2 Incremental and real-time GC

In the first systems with automatic memory management, the application program, or *mutator*[7], 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. For such applications, long GC pauses can be avoided by making the GC *incremental.*

Research within the field of incremental and real-time garbage collection has been going on since the late sixties. In the earliest attempts to implement non intrusive garbage collectors the GC work was split into a number of very small increments which were performed interleaved with the execution of the application [Bob68, Ste75, Wad76, DLM$^+$78, Bak78]. In order to guarantee progress of the garbage collector, a suitable number of increments of GC work are performed in connection with each memory allocation request, in proportion to the size of the request. An example of such an algorithm is Baker's algorithm [Bak78]. 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:

$$w \geq W_{max} \cdot \frac{a}{F_{min}} \tag{2.4}$$

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

[7]The term mutator comes from that, from the collector's point of view, the application is a process that changes, or mutates, the reference graph. In the sequel, the terms mutator and application will be used synonymously, when there is no risk of confusion. However, from the view of the underlying OS, a Java application includes both the mutator threads *and* the collector.

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 the mutator runs — thus causing interference. The problem of GC work always being performed when application threads run can be overcome by making the collector *concurrent*, i.e. assigning the GC work to a separate GC thread executing in parallel with the mutator threads. This is a strategy applied by a number of garbage collectors, e.g. the Appel-Ellis-Li collector [AEL88], but it has not been much used in real-time settings. Typically, in traditional concurrent collectors, no provision is made for guaranteeing that the collector keeps up with the allocation demands of the application.

Read and write barriers

When doing incremental garbage collection, the collector will operate on the heap while the mutator is potentially modifying the pointer graph. Therefore, mechanisms are required to ensure that mutator operations does not cause live objects to be missed by the collector, and that the operations of a moving collector does not cause dangling pointers in the mutator. Such mechanisms are called *read* and *write barriers*.

As barriers are preformed at every reference access it has been assumed that they, and especially read barriers, add a significant overhead. Therefore, much work has concentrated on developing algorithms that do not rely on barriers for synchronization between collector and mutator. However, a recent study found that, in many cases, the average overhead of both read and write barriers was small, and that the assumption is not always correct [BH04].

Read barriers are used in copying collectors, where reads of pointers to objects in fromspace are trapped in order to evacuate live objects and/or update pointers into fromspace to point to the new tospace copy.

Read barriers are also used in concurrent, moving collectors, like mark-compact, to ensure that pointer dereferences always return the current location of an object after it has been moved.

The current location of objects can be recorded in two ways. The first is the *forwarding pointer* approach typically used in copying collectors, where a field in the object header is used in fromspace objects to indicate their new, tospace, location. Another possibility is to use an *indirection table* outside the objects, where each object on the heap is pointed to from an entry in the table. The mutator do all accesses to objects via the indirection table, and thus, when an object is moved, only the table entry needs to be changed, and not all references to the object.

Write barriers are used in incremental or concurrent mark-sweep collectors to ensure that pointer updates during the mark phase cannot cause too few objects to be identified as live. Write barriers can be either of the *snapshot-at-the-beginning* or the *incremental update* type [Wil92, JL96], where the former is the more conservative of the two, ensuring that everything that was live at the start of the cycle will be retained. The latter works on the principle of preventing pointers to unmarked objects to be written into marked ones. An attempt to do so causes one of the objects to be (re)queued for marking.

2.3.3 Semi-concurrent GC scheduling

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 [Hen98]. 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 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 2.9 shows how the CPU time will be used in a system with one periodic high priority process and one low priority process.

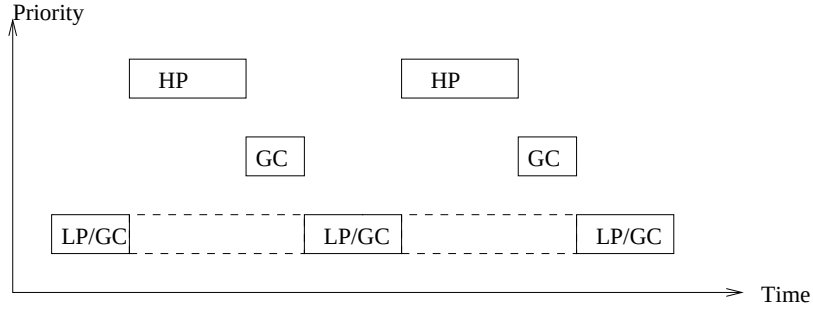


Figure 2.9: *Dividing the CPU time between processes. The system consists of one periodic high priority process (HP) and one low priority process (LP). Whenever a high priority process is suspended, and no other HP process is eligible for execution, the garbage collector (GC) is run. GC work is also 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 coarse 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 it is necessary to do a fair amount of scheduling analysis in order to tune the collector to a specific target platform.

2.3.4 Definitions

For clarity, this section and Figure 2.10 introduces the important terms used in the discussion of garbage collection scheduling. The operation of a GC is divided into GC cycles, and the time from the start (release) of a GC cycle to the end is called the GC cycle time, denoted T_{GC}. If nothing else is stated, the end time (deadline) of a GC cycle is equal to the release time of the following one. The execution time required to complete the GC work of one GC cycle is denoted C_{GC}. Scheduling of GC is aimed at avoiding out-of-memory situations, and the analysis is based on the amount of free memory, F, and the allocation rate, $\dot{a}$.

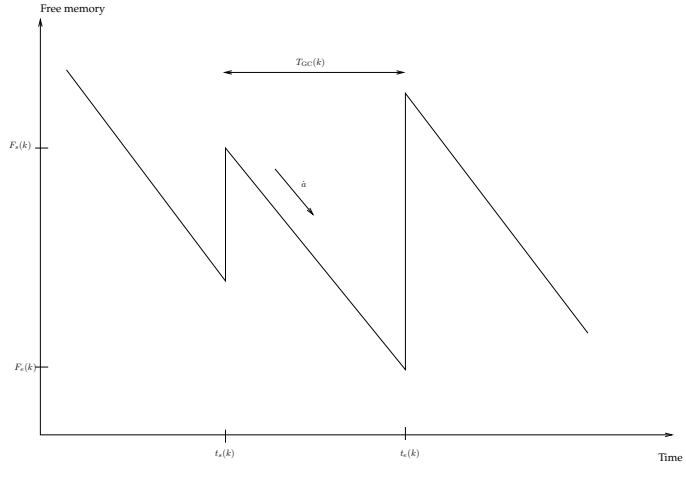


Figure 2.10: *Definitions used when discussing GC scheduling. The start and end time of a GC cycle is denoted t_s and t_e, respectively and $T_{\mathrm{GC}}(k) = t_e(k) - t_s(k)$ is the GC cycle time. $F_s(k) = F(t_s(k))$ and $F_e(k) = F(t_e(k))$ is the amount of free memory at the start (end) of GC cycle k, and $\dot{a}$ is the allocation rate.*

2.4 Real-time Java for embedded systems

Recently, Java has become more widely used in real-time applications and different solutions for developing and executing Java programs with timing requirements have been developed. We will now briefly review some of those.

Given a program, written in Java, there are basically two different alternatives for how to execute that program on the target platform. The first alternative is to compile the Java source code to byte code, and then have a, possibly very specialized, Java Virtual Machine (JVM) to execute the byte code representation. This is the interpreted solution (as required to be Java certified) used today for Internet programming, where the target computer type is not known at compile time. The second alternative is to compile the Java source code, or byte code, to native machine code for the intended target platform.

From the real-time garbage collection perspective, the differences between the two approaches are not significant, and the contributions of this thesis is applicable to both. The GC scheduling decisions are taken at a higher level, and is not dependent on instruction-level differences between platforms. There are also many similarities when GC implementation is considered. For instance, when a just-in-time (JIT) compiling JVM has compiled the byte codes to native code, this code is no different — to the GC — from code that was compiled ahead of time.

2.4.1 Real-time virtual machine

In virtual machines for real-time Java, the trade-off between predictability and performance becomes apparent. Just-in-time (JIT) compilation is very hard to combine with real-time demands, and using an interpreter typically has execution speeds 10 times slower than natively compiled code. To improve performance, some JVM (e.g. mackinac [Mac04]) use the JIT compiler to compile the application at initialization time. That however, comes at the cost of a significantly larger memory footprint. Also, the overhead of the JVM itself makes virtual-machine based solutions unsuitable for small embedded systems.

In order to speed up execution, reduce memory footprint or improve predictability, a number of hardware-assisted approaches to execution of Java byte-code have recently been developed. By using a co-processor to execute Java byte-codes, or by augmenting the instruction set of the processor with Java instructions, performance similar to that of native code can be achieved, without the overhead in time and space of a JIT compiler.

Most JVMs for the embedded market do not include real-time garbage collection, but rely on other mechanisms for memory management, like the scoped memory of the Real-time specification for Java (RTSJ) [B+01, Wel04].

With more and more large, high-performance computers used in control applications, the range of platforms which are referred to as "embedded" is vast. At the lower end of that range, virtual machines for embedded systems with only a few kilobytes of RAM can be found, including the Infinitesimal Virtual Machine (IVM) [Ive03] and SimpleRTJ [8]. For those systems, small memory footprint is the dominating design goal.

2.4.2 The Lund Java-based real-time platform

The Lund Java-based[9] Realtime Platform (LJRT) makes it possible to write hard real-time applications for small and medium sized embedded computers, in a portable way, using standard Java. The LJRT platform consists of two parts, the LJRT compiler and runtime system, and the LJRT class library.

The set of target systems considered include small (350 MHz PPC G3 with 32 MB ram) to very small (AVR μcontroller at 8MHz/32 kB RAM) embedded computers. Therefore, we prefer ahead-of-time compilation to using a JVM. One thing in common for almost all CPUs, is that there exists a C compiler with an appropriate back-end. In the interest of maintaining good portability while compiling Java to native code, C is used as the intermediate language; The Java front-end generates C code which, in turn, is compiled by a standard C compiler, as shown in Figure 2.11.

The compiler and run-time system part is made up of three loosely coupled components; the Java compiler, the garbage collector interface (GCI), and the run-time system. The Java compiler ([Nil04, NIEH04]) generates C code where all heap object accesses are made through the generic GCI ([IBE+02]) in order to provide an abstraction from the details of the different hard real-time garbage collector (GC) implementations ([Hen98, RH03]) that are part of the run-time system. To date, the run-time system has been ported to real-time Linux/RTAI/Xeno-

[8] `http://www.rtjcom.com`

[9] The term Java-based is due to the fact that our way of accomplishing a J2SE-compatible (any embedded program will run with the proper concurrency behaviour on any Java-enabled desktop) real-time Java platform is not compatible with the Java license conditions from Sun (we provide a real-time improved J2SE subset affecting the RTOS API without going via the JCP and without requiring a JVM). Thus, we may not call our free solution "Java", so we call it Java-based.

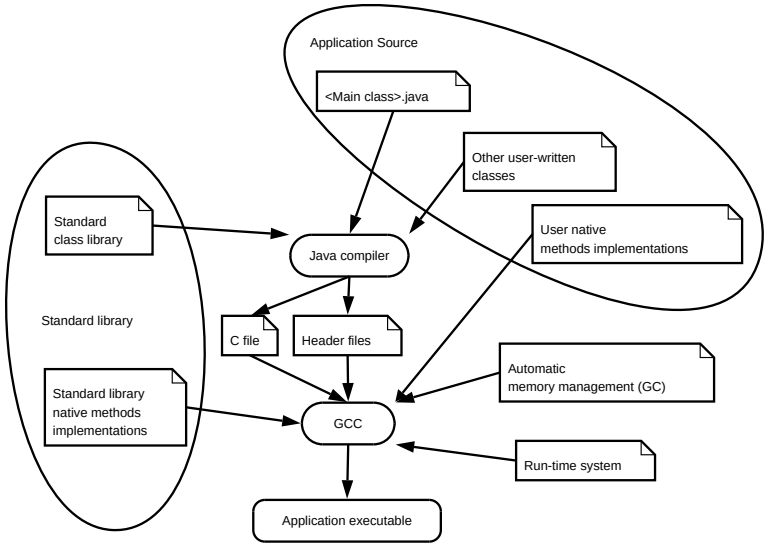


Figure 2.11: *LJRT compiler overview: The application Java source, together with the required classes from the standard library are translated to C by the Java compiler. That C code, any native method implementations provided by the user or the standard library, and run-time system code, GC, etc., is compiled and linked to produce the executable.*

mai with both user-space and kernel-space real-time threads on PowerPC and Intel, the STORK real-time kernel on PowerPC, a locally developed real-time kernel for the ATMEL AVR series of microcontrollers, and posix (for running in user-space, without hard real-time guarantees). Porting to a new RTOS is quite simple and requires writing a small number of native functions to interface with the RTOS system calls.

The LJRT class library is an open-source Java package containing classes for real-time threads, semaphores, monitors, mailboxes, etc. The LJRT library has both a pure Java implementation, allowing real-time applications implemented using the library to be executed on any JVM with proper concurrency behaviour, and native implementations, giving hard real-time performance on the target systems supported by the LJRT run-time system. The dual implementations are transparent to the user at the Java level, and the target-system specific features are automatically inserted through the LJRT compiler and run-time system.

Due to external requirements, we want to be able to use an off-the-shelf RTOS as well as external, legacy or automatically generated, C code. That, combined with using a standard C compiler as the back-end, means that we cannot rely on detailed assumptions on the behavior of the back-end C compiler or the thread scheduler, which makes implementation of a real-time GC more challenging. For instance, it means that any synchronization required between collector and application needs to be done explicitly. It also means that the generated C code must be written so that it ensures, in a portable way, that no back-end optimization causes interference with the GC. The challenges of implementing accurate real-time GC in an uncooperative environment is explored in Chapter 7.

2.4.3 Multi-stage deployment of control software

For future control systems, there is a strong need for tools and methods supporting the development and deployment of control software. To this end, we have proposed a method for developing hard real-time software based on the standard Java language and multi-stage deployment and verification towards the embedded platform [RNNH06]. As enabling technology, the LJRT platform is used, making it possible to develop embedded Java software on the desktop using standard software tools for implementation, testing, and verification, before deployment onto the embedded platform.

Development of embedded real-time software adds complexity compared to software development in general, as it typically includes writing, or interfacing to, proprietary hardware drivers (such as I/O), and cross compilation, resulting in platform-related problems. In order to mitigate these problems, it is desirable to separate platform concerns from application development. The presented method for development and deployment provides such a separation of concerns: The major part of the application can be implemented and its correctness in logical and concurrent behaviour verified, on the desktop, where building and execution is done using the standard Java SDK, and powerful development tools are readily available. In this stage, any process I/O, etc., is simulated. With the application working on the desktop, the move towards the embedded target system is done in steps where cross-compilation, drivers for I/O, and real-time requirements can be added and tested, one at a time.

While it is desirable to be able to do as much of the development and testing of an application as possible in a standard desktop environment, the subsequent port of the application to the hard real-time embedded system can require a large effort if done in an ad hoc manner. Therefore, we propose a method for doing the transition from desktop to target in a series of steps, where only one parameter is changed in each step, in order to facilitate verification of the different components, or identification of problems.

The fundamental principle is that the source code of the application should remain unchanged during all the stages of the deployment. What is changed, as the desktop application gradually is moved towards the embedded target, is, in turn, the class library, the compiler, the computer, the I/O drivers and the thread model. When the tools and the platform have been verified to work, it is possible to directly do the transition from the simulated environment on the desktop, to the target system. The major benefit of the intermediate steps is when things do not work, or when doing verification (or development) of the platform. The possibility of doing the deployment in several steps also makes it much easier to pinpoint at what stage of the deployment an error occurs and, hence, if the source of the error is in the application code, the tools, hardware drivers, or the operating system.

As a case study, a motion controller for the IRB-6 robot was developed. On the desktop, the application was run, in simulated time, on a standard JVM, with a virtual robot consisting of a simple dynamics model and Java3D visualization ([HN99]). On the real robot, the program was compiled to C code using the LJRT Java compiler and to native code with gcc. The target system was a Motorola MVME 2600-1 computer, with a 200 MHz PowerPC G3 CPU and the operating system was Linux/RTAI fusion[10], version 0.9.1. Figure 2.12 shows the real robot in the robot lab, and a screenshot of the virtual robot.

[10]Recently, the fusion branch of the RTAI project was moved into a separate project; Xenomai. RTAI fusion v0.9 corresponds roughly to Xenomai v 2.0.

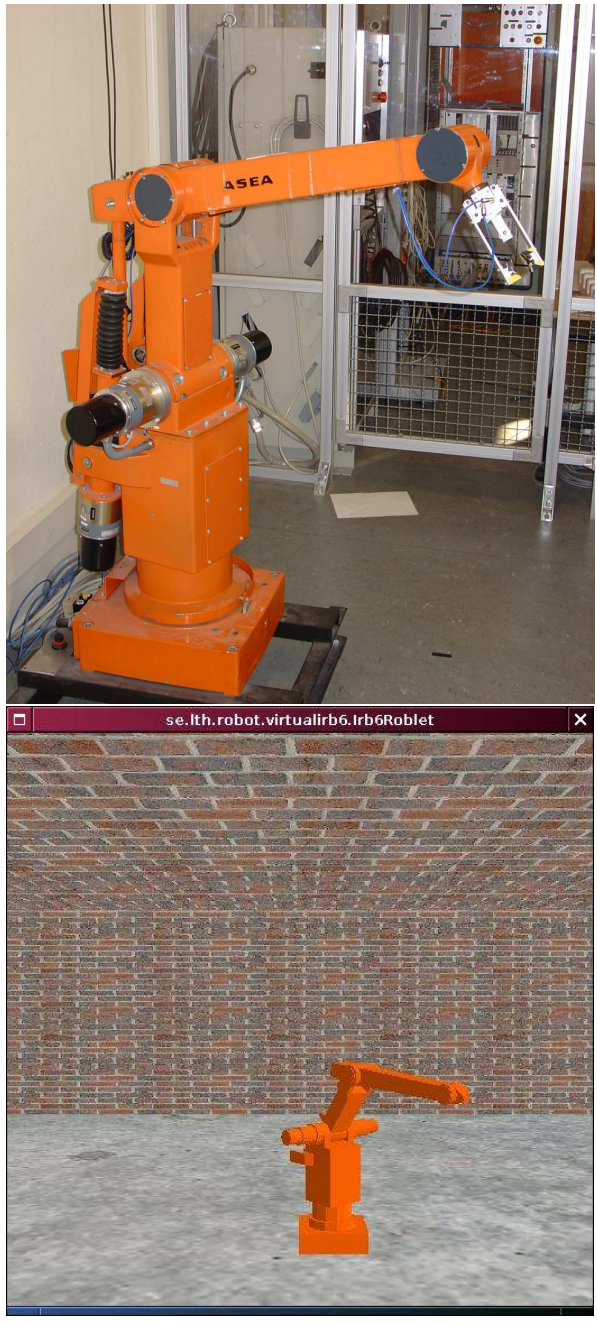


Figure 2.12: *The IRB-6 in the robot lab and its virtual counterpart.*

CHAPTER 3

TIME-TRIGGERED GARBAGE COLLECTION SCHEDULING

Traditionally, in order to ensure sufficient progress, 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 must be performed. This chapter presents a different approach where time, instead of allocation, is used as the trigger for GC work. 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 Section 3.2 an upper bound on the GC cycle time that ensures that new memory is always made available in time is formulated. Section 3.3 presents the problems associated with traditional metrics used to measure garbage collection work, and argues that time should be used as the unit for garbage collection work and that this is practically feasible. Section 3.4 discusses how the process scheduling strategy affects a time-triggered GC scheduler and it is 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.

3.1 Introduction

In [Rob02] the idea of time-based garbage collection scheduling 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, a hybrid approach was used, where the GC scheduling on the cycle level was time-based and the increments were scheduled using a traditional work metric in a fixed-priority scheduled system. This chapter presents time-triggered garbage collection more thoroughly, and we will see that having an explicit GC cycle time simplifies reasoning about more aspects of the memory system. It also mitigates or circumvents certain problems associated with real-time scheduling of an allocation-triggered GC. 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

[1] This is not the case for semi-concurrent scheduling, see Section 3.4.

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

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.

3.2 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 maximum allocation per period of thread j (a_j).

Lemma 1. *For a set of processes, $\mathbb{P}$, with, for each thread j, frequencies f_j, allocation requirements of a_j bytes per period and F bytes of memory available at the start of the GC cycle, it is guaranteed that the cycle will be completed before the available memory is exhausted if the GC cycle time, T_{GC}, satisfies*

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

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 necessary to cover the worst case schedule. A slightly stronger condition is

[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.

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

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

$$\begin{aligned}\sum_{j\in\mathbb{P}}(T_{GC}\cdot f_j+1)\cdot a_j &= \\ T_{GC}\sum_{j\in\mathbb{P}} f_j\cdot a_j+\sum_{j\in\mathbb{P}} a_j &\leq F \end{aligned} \tag{3.4}$$

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

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. $\square$

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. *For a set of processes with, for each thread j, frequencies f_j, allocation requirements of a_j bytes per period and a maximum total amount of live memory L_{max}, it is guaranteed that every GC cycle will be completed before the available memory is exhausted if the cycle time, T_{GC}, satisfies*

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

Proof. The theorem follows from lemmas 1 and 3. □

Remark. The term $\sum a_j$ is typically very small compared to the amount of memory available for allocation. (If not, heap occupancy is very high, a situation which is generally avoided, as it causes GC thrashing, increasing the CPU overhead of the GC.) Therefore, under normal circumstances, and for most practical reasons, it is safe to disregard this term, to get the simplified expression

$$T_{GC} \leq \frac{H - L_{max}}{2 \cdot \dot{a}} \tag{3.19}$$

where $\dot{a}$ is the total allocation rate of the mutator.

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.

Assume that at the start of the nth GC cycle there is $L_{max} = 50\%$ live memory (black), $G = 25\ \%$ floating garbage (dark gray) and $F_{min} = 25\ \%$ (white) available memory:

When the free memory has been allocated, the floating garbage and some of the objects that died during this cycle has been marked as garbage that will be reclaimed in this cycle (light gray) and some of the old objects become floating:

The GC cycle is concluded (i.e., the objects that are not to be reclaimed are compacted and a continuous area of available memory is formed): Note that during this cycle, we reclaimed more than F_{min}:

Therefore, we cannot use all the free memory during cycle $n + 1$ as that might result in less than F_{min} available memory in cycle $n + 2$. The solution is to reserve a part of the memory (striped) so that we only allocate $a_{max} = F_{min}$.

at the end of cycle $n + 1$:

the cycle is finished and the reserved memory is made available:

This cycle, we reclaimed less than F_{min}, but the amount of reclaimed memory + the reserved memory $= F_{min}$. Thus, the amount of available memory at the start of cycle $n + 2$ is F_{min} and our worst case assumptions hold.

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.*

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.

3.3 GC work calculation

In order to schedule an incremental or concurrent garbage collector so that it will finish at a certain time or after a certain amount of memory has been allocated, the amount of *garbage collection work* required to complete a GC cycle must be known. We will now examine how GC work can be expressed.

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. If a poor metric is used, a real-time algorithm may lose its real-time properties. For example, if we have a copying collector and use the number of evacuated objects as the work metric, we might reach a situation where we need to evacuate one more object to complete the current increment. However, this may — in the worst case — require us to scan all the remaining objects in the heap before we find a pointer that causes that last object to be evacuated. Thus, an unsuitable work metric causes the worst case amount of work, *in actual execution time*, of an increment *that is small according to the metric*, to be practically unbounded.

3.3.1 Traditional GC work metrics

For an allocation-triggered garbage collector, the minimum GC ratio, R_{min}, (in work units per allocated byte) that will ensure that the GC cycle finishes before the mutator runs out of memory is

$$R_{min} = \frac{W_{max}}{F_{min}}$$

where W_{max} is the worst case amount of work to complete a GC cycle and F_{min} is the worst case amount of available memory at the start of a cycle. Let the current GC ratio (R) be the ratio between performed work (W) and allocated memory (A):

$$R = \frac{W}{A}$$

In order to guarante that the GC finishes on time, we must ensure that the invariant

$$R \geq R_{min}$$

is satisfied at all times. Now, the problem is, how do we express, and measure, W? A common work metric for copying collectors is the evacuation pointer metric, i.e., use the amount of evacuated memory as a measure of performed GC work. Let ΔB denote the position of the evacuation pointer relative to the start of tospace (i.e., the amount of evacuated memory) 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} will be

$$W = \Delta B$$

$$W_{max} = E_{max}$$

Unfortunately, this metric doesn't 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, this metric may, in the worst case, cause an incremental collector to have a behaviour close to that of a batch GC.

This problem is described in [Hen98], 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 improved metric, as used in his semi-concurrent GC scheduling, is

$$W = \alpha \cdot roots + \beta \cdot \Delta S + \Delta B + \gamma \cdot \Delta P$$

$$W_{max} = \alpha \cdot roots_{max} + \beta \cdot E_{max} + E_{max} + \gamma \cdot M_{HP}$$

where S is the amount of scanned memory and P is the amount of initialized memory. The constants α, β and γ depend on the implementation of the algorithm and $roots_{max}$ and E_{max} depend on the application and these have to be manually tuned in order to make the discrepancy between the metric and the actual execution time as small as possible. For a compacting mark-sweep collector, a similar GC work metric looks as follows

$$W = \alpha \cdot (roots + mark) + \beta \cdot sweep + \gamma \cdot compact$$

$$W_{max} = \alpha \cdot (roots_{max} + live_{max}) + \beta \cdot heapsize + \gamma \cdot live_{max}$$

On the other end of the scale are the concurrent algorithms that have a separate GC thread which performs garbage collection in parallel with the mutator. In this case, the collector thread is run without synchronization with the mutator (in the sense that it does GC work until the cycle is complete and then waits for another cycle to be triggered.).

3.3.2 Using time as the GC work metric

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].

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.

[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.

3.4 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 3.2, 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 runtime 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 [Hen98].

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.

3.4.1 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 must maintain the invariant

$$\sum w \geq W_{max} \cdot \frac{t - t_{cycle\ start}}{T_{GC}} \tag{3.20}$$

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 (2.4) on page 31 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.

3.4.2 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 di-

viding 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.

3.5 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

Worst case analysis is, in the general case, difficult even for relatively small programs and for a concurrent garbage collector it is even harder, as the execution time of the garbage collector not only depends on the GC implementation and application code per se, 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 depends 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, leading to unacceptably low CPU utilization. Using feedback control, on the other hand, makes it possible to exploit varying resource utilization among the application threads, allowing better overall utilization of both CPU and memory. If the CPU overhead of memory management is made explicit, in a feedback scheduling system, that information can be measured at run-time and taken into account when scheduling the application threads[1].

We will now investigate how a time-triggered GC can be made auto-tuning by estimating the scheduling parameters of the GC thread at run-time. Section 4.1 gives an introduction to the problem to motivate the work, and gives an overview of the proposed approach. In order to schedule a task, two parameters are needed; its deadline and its execution time. Section 4.2 shows how the cycle time can be automatically tuned and Section 4.3 discusses how the amount of CPU time required to complete a GC cycle can be estimated.

[1] An approach to incorporating an auto-tuning GC into a feedback scheduler is suggested in Chapter 6.

4.1 Introduction

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 requires a great engineering effort and is 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. This implies that using an adaptive strategy may result in a more robust system compared to a manually tuned system where the worst-case estimates have been found using measurements and a safety margin.

The proposed adaptive garbage collection scheduling model consists of two 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, either directly, as in the EDF case, or indirectly, when using RMS 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 in the on-line schedulability analysis required to guarantee that the system remains schedulable.

In a system with garbage collection, allocations can be measured continuously whereas measurements of the heap state are only available after the completion of a GC cycle. Therefore, the proposed approach has the structure sketched in Figure 4.1. The scheduling parameters are tuned based on measurements of the amount of available memory and the allocation rate. The work function describing how the execution time of the GC depends on the heap state is based on previous measurements of the heap state and GC execution time.

In the cycle time tuning, a black-box view on the application is used; the estimates do not depend on any information about the application other than the allocation rate, which can be measured directly. 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 4.3 discusses both a black box and a clear box approach to garbage collection work estimation.

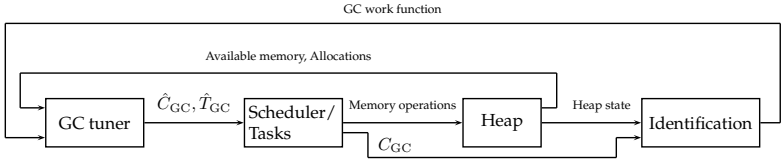


Figure 4.1: *Block diagram of an adaptive GC. Based on measurements of the amount of available memory, the allocation rate of the application, the heap state and the previous execution of GC, the cycle time and execution time of the GC is estimated.*

4.2 Automatic GC cycle time tuning

As we have seen in Chapter 3, a GC cycle length that ensures that the application never runs out of memory can be calculated at design-time, if the allocation requirements of the (high priority) mutator threads are known. If that 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 automatically tune the GC scheduling parameters on-line.

When doing on-line tuning without any information about the application, the fundamental problem is that the amount of live memory and floating garbage is not known and must be estimated in a safe and robust way. Section 4.2.1 examines, in more detail, the model for how the GC cycle time can be automatically tuned without any a priori information about the application. Section 4.2.2 investigates how the GC scheduling can be improved if some information about the behaviour of the application is available; for instance, through feed-forward of mode changes. The GC cycle length depends on the allocation rate, and as

allocations typically are bursty, the allocation rate estimation must be done carefully, which is discussed in Section 4.2.3. Feed-forward from the mutator to the GC scheduler is discussed in Section 4.2.4.

4.2.1 Application-independent auto-tuning

The fundamental requirement on the GC cycle time is that each GC cycle must finish before the application runs out of memory. In an on-line GC tuner, that can be achieved by calculating or measuring the allocation rate ($\dot{a}$) of the application and extrapolating at which time all the currently remaining free memory (F) will have been allocated — the deadline of the current GC cycle.

Let t_s denote the start(release) time of the current cycle and $t_e = D_{\mathrm{GC}}$ the deadline of the GC cycle. The GC cycle must end before the time when all free memory will have been allocated. Therefore, at time t ; $t_s \leq t < t_e$, assuming that $\dot{a}$ is constant, we can extrapolate when all memory has been allocated, and we get the constraint

$$t_e \leq t + \frac{F(t)}{\dot{a}} \tag{4.1}$$

which gives the cycle time

$$T_{GC} = t_e - t_s \leq t + \frac{F(t)}{\dot{a}} - t_s \tag{4.2}$$

The simple model of Equation (4.2) will work if the same amount of memory is reclaimed in each GC cycle but it suffers from the same problems with floating garbage as the fixed deadline case discussed in Chapter 3, although the symptoms are a bit different. With a fixed deadline, the system might run out of memory if the GC cycle time is too long. In an adaptive system where the cycle time is tuned to ensure that this does not happen, the problem is that the system might become unschedulable. One example of this encountered during 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[2]. Then, the following cycle will have to be very short and we get a memory trace like the one shown in Figure 4.2. 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

[2]Variation in the amount of floating garbage is mainly a concern when using an incremental-update GC. The conservatism of snapshot-at-the-beginning collectors will give more floating garbage but less variations.

of GC work is roughly the same[3] in all cycles, but it has to be done in a much shorter time in the short cycles.

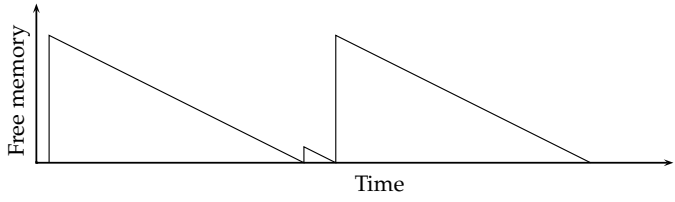


Figure 4.2: *Example of a very short GC cycle caused by large amounts of floating garbage.*

In order to handle the worst case amount of floating garbage, memory must be reserved so that the allocations during the next cycle can be satisfied even if no objects are reclaimed during the current cycle. I.e., the GC cycle must end before all available memory has been allocated.

Theorem 2. *Let $\hat{\dot{a}}(t)$ be an estimate of an unknown but constant allocation rate, $\dot{a}$, such that $\hat{\dot{a}}(t) \geq \dot{a}$. Then, for $t_s(k) \leq t < t_e(k)$, the GC cycle will be completed before the available memory is exhausted if the cycle time, T_{GC}, satisfies*

$$\hat{T}_{\mathrm{GC}}(t) = \frac{1}{2}\left(t + \frac{F(t)}{\hat{\dot{a}}(t)} - t_s(k)\right) \tag{4.3}$$

Proof. Let $\dot{a}(k)$ be the allocation rate during GC cycle k. Then, the amount of memory allocated during cycle k is $a(k) = T_{\mathrm{GC}} \cdot \dot{a}(k)$. In the worst case, no memory is reclaimed during cycle k, so $a(k+1)$ bytes must be reserved for the following cycle in order to satisfy all allocations. I.e., the requirement is that

$$F_s(k+1) \geq T_{\mathrm{GC}} \cdot \dot{a}(k+1) \tag{4.4}$$

During cycle k, i.e., for $t_s(k) \leq t < t_e(k)$, the amount of memory available at the start of cycle $k+1$ is

[3]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.

$$F_s(k+1) \geq F(t) - (t_e(k) - t)\,\dot{a}(k) \tag{4.5}$$

with equality in the worst case, that no memory is reclaimed during cycle k. Using the equalities in (4.4) and (4.5) we get

$$T_{\mathrm{GC}} \cdot \dot{a}(k+1) = F(t) - (t_e(k) - t))\dot{a}(k) \tag{4.6}$$

$$\therefore t_e(k) = t + \frac{F(t) - T_{\mathrm{GC}} \cdot \dot{a}(k+1))}{\dot{a}(k)} \tag{4.7}$$

Thus, the GC cycle time estimate is

$$T_{\mathrm{GC}} = t_e(k) - t_s(k) = t + \frac{F(t) - T_{\mathrm{GC}} \cdot \dot{a}(k+1))}{\dot{a}(k)} - t_s(k) \tag{4.8}$$

which can be rearranged as

$$T_{\mathrm{GC}} = \frac{F(t) + (t - t_s(k))\,\dot{a}(k)}{\dot{a}(k) + \dot{a}(k+1)} \tag{4.9}$$

If the allocation rate is constant, i.e., $\dot{a}(k+1) = \dot{a}(k)$, we get (4.3). □

If the allocation rate is constant, this means that we should reserve half of the available memory at the start of the current cycle for the allocations during the next GC cycle. Doing so guarantees[4] that we can handle the worst case, when all the objects that die during a cycle become floating garbage and will not be reclaimed until at the end of the next GC cycle. Figure 4.3 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.

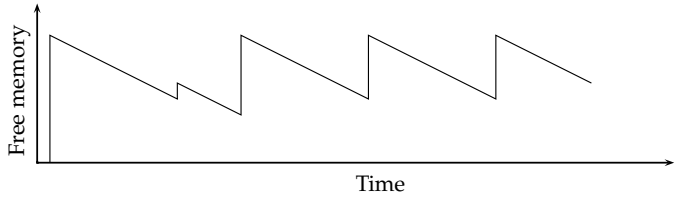


Figure 4.3: *Example of how reserving memory for the next cycle mitigates the problems of floating garbage depicted in Figure 4.2.*

As GC cycles are shortened, the number of GC cycles increase and consequently the incurred GC overhead increases. However, as we do

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

not use all of the heap, the additional overhead is not as big as it would seem. Also, only allocating at most half of the available memory each GC cycle might seem wasteful, but this is the price we pay for incrementality. It should, however, be noted 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.

In (4.3), it is assumed that the allocation rate is constant. For a typical control system with a number of periodic threads running the same control algorithm every sample, that is a reasonable assumption, and in experiments, the GC cycle time estimates have been stable and accurate. Also, note that the assumption that $\dot{a}$ is constant only means that T_{GC} is chosen to ensure that the allocations can be satisfied at the current rate. If the allocation rate changes, the auto-tuner will change the T_{GC} estimate according to the new allocation rate, to ensure that all allocations can be satisfied under the assumption that allocations will continue at the new rate.

4.2.2 Using information about the application

If the GC cycle times are tuned according to Equation (4.3), the risk of running out of memory due to floating garbage is reduced, but the cycle times, and thus the CPU utilization of the GC will vary if there are big variations in the amount of floating garbage. In particular, the GC cycle time estimates will, in the average case, be quite conservative. This is due to the fact that if the GC cycle time tuner has no information about the behaviour of the mutator, it cannot differentiate between an unusually large amount of floating garbage and an actual increase in the amount of live memory, where the former should not affect the GC cycle time, but the latter should. Therefore, under the proposed strategy, it must always ensure that no more than half of the available memory at the start of a cycle is used during that cycle. That means that in the extreme case that nearly all of the objects that die during a cycle becomes floating garbage, the cycle time estimate will be halved, as shown in Figure 4.3. That is, of course, better than without any reservation strategy, but still unnecessarily conservative.

Based on this observation, we will now see how having information about the behaviour of the mutator can improve the GC cycle time estimates. In common special cases, additional information about the state of the mutator and memory system allows using a less conservative GC cycle time estimate. Such special cases include when the application

is known to be in steady-state, and allocation and release of large data structures, including creation and deletion of processes.

If the system is known to be in steady-state, the amount of live memory is constant[5]. Then, the variations in available memory at the start of GC cycles are due to variations in floating garbage. The example in Figure 4.4 shows how this information can be used to avoid unnecessary changes to the GC cycle time.

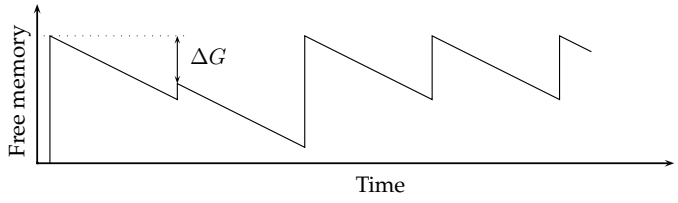


Figure 4.4: *Example of how information about the amount of floating garbage allows a less conservative GC scheduling strategy. If we know that the system is in steady state, the difference in free memory at the start of the cycles is due to floating garbage. Thus, during the second cycle, we know that at least ΔG bytes will be made available after the cycle and we can allow allocation of more than half of the available memory.*

Similarly, information about changes in the amount of live memory can be used. While performing worst case live memory analysis in the general case is very difficult, programmers — especially when developing real-time and embedded systems — will have a reasonably good idea about what persistent data structures each process uses. If a mode change requires some data structure to be allocated or causes some other data structure to go out of scope, this is typically known at design time. By informing the GC tuner about this, it can react to the changes in the amount of live memory sooner and in a more accurate way [6]. One special case is when a process is created, the amount of live memory will increase at a well-defined point in time. Conversely, when a process dies, the amount of live memory will decrease. Typically, a process has a set of persistent objects. E.g., in a control system, a process will typically create a set of objects for inputs, outputs, and control algorithms,

[5] In practice, that is merely an approximation, as a fraction of the allocated objects are used for temporary results and not for persistent data, which adds small, high-frequent, variations to the amount of live memory. Still, if the GC cycle time is much longer than the period times of mutator processes, the impact of temporary objects will be small.

[6] Having a hint about object liveness is much less dangerous than explicit (manual) deallocation; the former only affects the scheduling of garbage collection, whereas the latter may cause dangling pointers and memory leaks.

and the size of these may be found by compile-time analysis, especially if doing whole-system compilation.

Just as in the case where the system was known to be in steady state, if the amount of live memory has changed by a known amount, similar reasoning can be used, with the addition of taking the change in live memory into account: If there is less memory available at the start of a GC cycle than at the start of the previous one, the sum of floating garbage and live memory has changed, and if the change in live memory is known, the change in floating garbage can be calculated.

From the preceding discussion we see that information about changes in the amount of live memory, or knowing that the application is in steady state, makes it possible to estimate the amount of floating garbage. With that information, the GC cycle time estimates can be less conservative, allowing more uniform resource utilization. We will now formalize that idea. In order to reason about differences in the amount of memory reclaimed in different GC cycles, the amount of free memory just after the reclaimed memory has been made available is used. For convenience and clarity of the presentation, let $F_s(k) = F(t_s(k))$ denote the amount of free memory at the start of GC cycle k.

If information about the state of the memory system is available, it is possible to generalize Theorem 2 slightly.

Theorem 2a. *Let $\hat{\dot{a}}(t)$ be an estimate of an unknown but constant allocation rate, $\dot{a}$, such that $\hat{\dot{a}}(t) \geq \dot{a}$, and $\Delta F_{ff}(k) \geq 0$ an amount of memory that is known to be reclaimed during GC cycle k. For $t_s(k) \leq t < t_e(k)$, the GC cycle will be completed before the available memory is exhausted if the cycle time, T_{GC}, satisfies*

$$\hat{T}_{\mathrm{GC}}(t) = \frac{1}{2}\left(t + \frac{F(t) + \min(\Delta F_{ff}(k), F_s(k))}{\hat{\dot{a}}(t)} - t_s(k)\right) \tag{4.10}$$

Proof. If at least $\Delta F_{ff}(k)$ will be reclaimed during cycle k, then

$$F_s(k+1) \geq F(t) - (t_e(k) - t)\,\dot{a} + \Delta F_{ff}(k) \tag{4.11}$$

In order to satisfy the allocations of cycle $k+1$, it must hold that

$$F_s(k+1) \geq T_{\mathrm{GC}} \cdot \dot{a} \tag{4.12}$$

In analog with the proof of Theorem 2, that gives

$$T_{\mathrm{GC}} = \frac{1}{2}\left(t + \frac{F(t) + \Delta F_{ff}(k))}{\dot{a}} - t_s(k)\right) \tag{4.13}$$

However, as the GC cycle must still end before the available memory is exhausted, another condition is that

$$F(t) - (t_e(k) - t)\dot{a} \geq 0 \tag{4.14}$$

which, by using $t_e(k) = T_{\mathrm{GC}} + t_s(k)$ and reorganizing gives

$$F(t) - \Delta F_{ff}(k) + (t - t_s(k))\dot{a} \geq 0 \tag{4.15}$$

But if $\dot{a}$ is constant, $F(t) + (t - t_s(k))\dot{a} = F_s(k)$ and thus (4.13) is safe if $\Delta F_{ff}(k) \leq F_s(k)$.

Otherwise, as any reclaimed memory will not be made available until in the next GC cycle, the compensated GC cycle will be too long, and the system will run out of memory. Therefore, if $\Delta F_{ff}(k) > F_s(k)$, the compensating term must be limited. (4.2) is an upper bound on the feasible GC cycle times. I.e., the GC cycle time must satisfy the constraint

$$T_{\mathrm{GC}} = \frac{1}{2}\left(t + \frac{F(t) + X}{\dot{a}} - t_s(k)\right) \leq t + \frac{F(t)}{\dot{a}} - t_s(k) \tag{4.16}$$

where X is the compensating term. Reorganizing gives

$$X \leq F(t) + (t - t_s)\,\dot{a} \tag{4.17}$$

and, for a constant allocation rate, that is equivalent to

$$X \leq F_s(k) \tag{4.18}$$

which obviously is satisfied for

$$X = \min(\Delta F_{ff}(k), F_s(k)) \tag{4.19}$$

Thus, the amount of memory reserved for cycle $k + 1$ can safely be reduced by $\min(\Delta F_{ff}(k), F_s(k))$. □

Given information about the behaviour of the application, the amount of floating garbage can be estimated, and Theorem 2a can be applied in order to reduce conservatism in the GC cycle time.

Theorem 3. *Let $\Delta L(k)$ be the net change in live memory during GC cycle k. For $t_s(k) \leq t < t_e(k)$, a safe upper bound on the GC cycle time is*

$$\hat{T}_{\mathrm{GC}}(t) = \begin{cases} \dfrac{t + \frac{F(t) + \min(\Delta G(k-1), F_s(k))}{\dot{a}(t)} - t_s}{2} & ; \quad \Delta G(k-1) > 0 \\ \dfrac{t + \frac{F(t) +}{\dot{a}(t)} - t_s}{2} & ; \quad \text{otherwise} \end{cases} \tag{4.20}$$

where

$$\Delta G(k) = F_s(k) - F_s(k+1) - \Delta L(k) \tag{4.21}$$

Proof. First, consider the change in floating garbage. The heap contains live objects, garbage, and free memory: $H = L+G+F$. As the heapsize, H, is constant, comparing the heap state at the start of GC cycles k and $k+1$, respectively, gives

$$L_s(k) + G_s(k) + F_s(k) = L_s(k+1) + G_s(k+1) + F_s(k+1) \quad (4.22)$$

Introducing the symbols ΔL and ΔG and rearranging gives (4.21). Now, for the GC cycle time:

(i) If $\Delta G(k-1) > 0$, the total amount of floating garbage in cycle $k-1$ must have been at least $\Delta G(k-1)$. As floating garbage will be reclaimed in the following cycle, the amount of memory made available before the start of cycle $k+1$ must also be greater than or equal to $\Delta G(k-1)$, and the result follows from Theorem 2a.

(ii) If $\Delta G(k-1) \leq 0$, nothing is known about the absolute amount of floating garbage, and T_{GC} must be estimated according to Theorem 2. □

Remark. Equation (4.21) estimates the amount of floating garbage based on knowledge about changes in the amount of live memory and the difference in free memory in two GC cycles. That estimate can be improved by using a longer time horizon: If the system is in steady state, a high-water mark of the amount of free memory at the start of a cycle since the system entered steady state gives a minimum for the sum of live and floating objects. Thus, by comparing the current amount of free memory with the high-water mark, a less conservative estimate of the amount of floating garbage is obtained. I.e., if the system has been in steady-state since cycle $j < k$, (4.21) can be replaced by

$$\Delta G(k) = \max_{i \in [j,k]} \{F_s(i) - F_s(k+1)\} \quad (4.23)$$

If feed-forward information about changes in the live memory amount is available, the high-water mark must be adjusted correspondingly.

4.2.3 Estimating allocation rate

In the preceding discussion, it was assumed that the allocation rate, or a conservative estimate of it, was available. We will now briefly examine some properties of allocation rate measurement and how such an estimate can be obtained.

As allocations are discrete events, there is, by definition, no instantaneous allocation rate that can be measured, so for any discussion, an

average allocation rate must be used. Allocations are carried out at arbitrary places in the mutator code, so on a short timescale the allocation rate will vary with very high frequency. The GC cycle tuner will typically run at a much slower rate than this, and therefore the allocation rate measurements can be viewed as slow sampling of a signal with high-frequency components. Thus, there is a risk that those high-frequency components introduce low-frequency noise into the allocation rate measurement, through aliasing. Also, if there are multiple processes, and the variations happen to be aliased into frequencies that are close, the effect can be exagerrated by interference beating [AW97].

The allocation rate estimate is used to determine the GC cycle time, so the estimate must not be too low as that may cause the application to run out of memory. Therefore, using simple averaging or a normal low-pass filter is not suitable, as it may smooth out steps in the allocation rate, leading to temporary under-estimation. One method that is simple to implement and has proven to work well in practical experiments is to periodically measure the amount of allocated memory, and filter by using the maximum (averaged over a certain time window) allocation rate, combined with a forgetting factor for the max value, to give suitable responsiveness to changes.

By using feed-forward, the estimation can be improved. If threads are periodic, and execute the same code in each invocation, the allocation rate (expressed as allocations per period) can be measured exactly by simply recording the allocations performed by each thread from one release to the next. In that case, there will be no aliasing, as the sampling is synchronized with the allocations. By measuring the allocation rate seperately for each periodic thread, interference effects are eliminated. If there are small variations, that can be detected and handled by some filtering (max + a forgetting factor). In addition, if the memory manager is informed about changes to the allocation rate, measurements can be low-pass filtered in order to reduce noise, while still reacting quickly to actual changes. Also, if it is known that a particular allocation is a one-time occurence (e.g., allocating a large persistent data structure at start-up), it should not affect the allocation rate estimate (although it may affect the amount of live memory).

4.2.4 Feed-forward from the application

The results in Section 4.2.2 are based on having information about the operation of the application. In order to satisfy that requirement, this section sketches a set of feed-forward operations for both qualitative and quantitative information about the memory usage of the mutator.

Qualitative feed-forward

The feed-forward has to be provided by the mutator code, where the feed-forward instructions have to be inserted manually or, perhaps, automatically by a tool. In order to be practically feasible, the amount of analysis required for finding the feed-forward information must be kept at a minimum, and therefore, a model requiring only qualitative information about the mutator behaviour is desirable. As we have seen, the simple information about whether the application is in steady state or not is quite valuable to the on-line auto-tuner.

If all threads are in steady state, the amount of live memory is constant (i.e., $\Delta L = 0$), and Theorem 3 may be used. If one or more mutator threads are in a transient state, the GC cycle time estimate must be done according to Theorem 2 until the finish of the GC cycle *after* the one where all threads had returned to steady state. Conversely, during the GC cycle when a thread enters the transient state, the amount of floating garbage from the cycle before is known, and is known to be reclaimed at the end of the cycle, and the GC cycle time may be calculated using Theorem 3.

Mode changes can cause transients in the memory usage pattern of an application. As discussed, the only continously available measurements the GC auto-tuner can use is allocation rate and amount of allocated memory. Large one-time allocations e.g. at the start-up of a thread or at a mode change may cause spikes in the allocation rate measurement, leading to changes in the GC scheduling. Such effects can be mitigated with information that a certain memory allocation is a one-time occurrence.

If the GC scheduler knows that a thread is executing periodically, that can be used to improve allocation rate estimations. That information is often available, as many real-time operating systems has a special type of thread or process for periodic tasks. Additionally, threads which are not periodic in the usual real-time programming sense, may execute periodically. One example of such a thread is a controller in a distributed control system, receiving measurements from a remote node, over a network, as sketched in Figure 4.5. In the code, the controller is not a periodic thread, it is simply blocked waiting for the next package on the network. However, if the sampling thread on the remote node is periodic, network packages will arrive periodically and the controller will effectively be periodic. The period time can either be measured, by recording release times, or explicitly fed forward from the application.

```
while(!interrupted()) {
    Sample s = receiveSample(); // Blocking call
    Control c = compute(s);
    output(c);
}
```

Figure 4.5: *Simple example of a main loop of a controller thread in a distributed control system. This code is not periodic* per se. *However, if samples arrive at a fixed rate, it will be* effectively *periodic.*

Quantitative feed-forward

For a thread that is known to be periodic, the period time is also often known, either at design time, or — in a feedback scheduling system — at run-time. For effectively periodic threads, where the period time isn't known locally, it is useful to explicitly state this information which is available somwhere in the system. Also, if period times are changed at run-time, it is better to feed forward this information at the time of the change than waiting for it to show up in measurements.

If a mode change is known to affect the amount of live memory, that information can be used to improve the GC scheduling, as in Theorem 3. As discussed, in embedded systems development, the programmer is often required to know the memory requirements of an application to ensure that there is enough memory in the system. The memory usage figures could also be obtained by a worst case analysis tool. Using the approach taken e.g. in [Per99], annotations can be used to perform the analysis for the different modes of operation.

4.3 GC workload prediction

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 GC increments depends on it[7]. Also, in feedback scheduling systems, on-line schedulability analysis is performed and the allowed CPU time utilization of the application threads is tuned to keep the total requested CPU utilization at the setpoint. Therefore, in such systems, it must be possible to determine how much CPU time that is required in order to complete

[7]However, for the real-time performance of high priority threads, it is enough that it is conservative; over-estimating the CPU requirement of the GC only leads to (temporary) starvation of background threads.

a GC cycle. 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 mutator, 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.

Thus, in an adaptive system, the role of the workload estimation is to feed-forward information about changes in required CPU utilization to the scheduler, so that any necessary change in scheduling parameters may be done before the measured CPU utilization gets too high. Also, in an adaptive system, there are no absolute guarantees, but rather a trade-off between safety and performance, and GC work prediction can be more or less conservative. Techniques for producing both tight and conservative C_{GC} estimates will be discussed.

In many cases, the occasional under-estimation is not a problem; As stated, feedback scheduling works on the principle of measuring actual CPU utilization and changing scheduling parameters in order to handle overload conditions, and is therefore inherently robust to overload. This is reinforced by the fact that the T_{GC} estimates are based on worst case assumptions and therefore usually are conservative, giving some slack in the schedule. Conversely, if a conservative C_{GC} estimate is used, the resulting slack in the schedule is not wasted, but can be utilized by the mutator if the FBS is aware of when the GC is running and when it is idle.

4.3.1 Black box estimation

A black box model doesn't use 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.

A simple scheme which has been experienced to work fairly well in practice is to estimate the GC cycle execution time with the highest value during the last n cycles. Another alternative is to use e.g. a moving average filter, but that has a greater risk of under-estimating the execution time, where using the max value tends to be conservative.

The main drawback of any such approach 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.

4.3.2 Clear box prediction

In a clear box approach, the principle is to measure a number of parameters of the memory system, and, using some automatic system identification technique, determine how they affect the execution time of the GC. That requires a more detailed interface between the GC scheduler and the memory management system.

In order to predict the amount of GC work, a GC work metric is required, expressing GC work as a function of the state of the heap

$$C_{\mathrm{GC}} = f(S_h). \tag{4.24}$$

Given the structure of GC algorithms, 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 mark all objects is proportional to the number of live objects, the time required to evacuate live objects depends on the size of the live memory, initialization of memory depends on the amount of dead memory, etc. Thus, an approximation of the GC workload can be expressed as

$$C_{\mathrm{GC}} = K\ S_h \tag{4.25}$$

for some vector K, which is identified on-line. Given a function f, or coefficient vector K, 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 automatically tuned to fit different GC algorithms.

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

Measure and predict the heap state: In its most simple form, only the amount of available memory is measured. A more detailed model would take into account 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.).

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. Some operating systems also provide execution time statistics.

Identify a GC work function: In order to predict the amount of GC work required to complete a GC cycle, a function from heap state to GC execution time has to be identified. If a linear model is assumed, the problem becomes on-line estimation of the elements of K, given past measurements of C_{GC} and S_h, which can be done e.g. with a recursive least squares algorithm [AW89].

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 by inserting the predicted heap state into the identified work function.

Measuring and predicting heap state and predicting C_{GC} will now be discussed in more detail.

Measuring and predicting heap state

Of course, 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, but may float for one cycle, which leads us to the following abstract representation of the heap state:

$$S_h = \begin{bmatrix} \#\ live\ objects \\ \#\ live\ bytes \\ \#\ dead\ objects \\ \#\ dead\ bytes \\ \#\ floating\ objects \\ \#\ floating\ bytes \end{bmatrix} \tag{4.26}$$

A problem with garbage collection is that some aspects of the heap state, like for instance the amount of live or dead memory, can only be observed at the end of GC cycles. Even worse, with an incremental GC, it is not possible to distinguish between live memory and floating garbage. Therefore, the heap state cannot be measured directly, but must be calculated based on what can be measured. It is possible to formulate a dynamic system that, under certain assumptions, is observable. However, for a system with n states, it takes at least n samples for the error to reach zero. In this case, samples equals GC cycles, meaning that the model would be quite slow. Combined with the noisy measurements (e.g. due to floating garbage) such a detailed model would be problematic in practice.

Also, (4.26) fails to capture two important factors; the actual placement of the objects on the heap, and the distribution of references in objects. 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. The reference content of objects affects the time required to trace the live object graph, with the extremes being data arrays at one end of the spectrum, and reference arrays at the other.

Therefore, using (4.26) as an abstraction of the heap state and attempting to predict it by simulating a dynamic system appears problematic for two reasons. Using an observer to reconstruct many states limits how quickly the model can react to changes, and the approximations done still leaves out important aspects that affect the GC workload.

We need some way of predicting S_h based on quantities that can be measured. Therefore, the approach taken here is to use a simplified heap state representation, only using the number of live (L) and dead (D) bytes and not taking object sizes into account[8].

$$S_h = \begin{bmatrix} L \\ D \end{bmatrix} \tag{4.27}$$

Then, in principle, the heap state can be predicted by finding the probability of a memory cell being live or dead, respectively, and applying that to the total amount of allocated memory (A):

$$\hat{L} = P(Live) \cdot A \tag{4.28}$$

and

$$\hat{D} = P(Dead) \cdot A \tag{4.29}$$

That has the advantage that while L and D cannot be observed directly, A can be measured at any time. However, what is interesting for predicting $C_{\mathrm{GC}}(k)$ is a prediction of the amount of allocated memory at the end of the GC cycle, $A_e(k)$, which can be predicted by extrapolation similar to that in the T_{GC} calculation:

$$\hat{A}_e(k) = A_s(k) + T_{\mathrm{GC}}(k) \cdot \dot{a} \tag{4.30}$$

The prediction of L and D is then given by inserting that value into (4.28) and (4.29).

[8]The terms live and dead memory are actually not very accurate in this context; what is interesting for the amount of GC work is what the garbage collector *thinks* is live and dead memory. In this presentation, the terms live and dead should be understood as synonyms for *retained* and *reclaimed*, respectively.

Now, $P(Live)$ and $P(Dead)$ must be found. Excluding startup, typical embedded or other long-running programs with a well-defined set of tasks can be expected to behave quite similarly from one GC cycle to the next. For such systems, the fraction of live (dead) memory in the previous GC cycle(s) can be used: $P(Live) = \frac{L}{A}$. Robustness against variations in live and dead memory due to e.g. floating garbage can be achieved by adding low pass filtering using the maximum, median or mean observed value, and responsiveness to actual change by using a forgetting factor for reducing the weight of old measurements.

A potential problem with using only the amount of live and dead memory is that if the GC work function have been identified on-line, based on past measurements of L, D, and C_{GC}, there is no guarantee that the function will be valid if the distribution of objects changes, as if does not take the number of objects, pointer density, or placement, into account. Therefore, an extreme change in object distribution like, e.g., from the heap being dominated by a highly connected linked structure of small nodes to consisting mainly of huge data arrays, might cause a large error in the work estimate, until the work function identification has had time to react. In practice, this is unlikely to be a problem. Firstly, while mode changes often occur, they are seldom as drastic as that, and with many threads, effects are likely to even out. Secondly, previous work has shown that e.g. the variation in pointer density and fraction of non-null pointers between the different SPECjvm benchmarks is quite low [BCR03a].

Predicting GC work

Now, we need to put it all together into a prediction of the amount of CPU time required to complete a GC cycle. With the simple heap state model, the GC work function is

$$C_{\mathrm{GC}}(L, D) = \alpha\, L + \beta\, D \tag{4.31}$$

where the coefficients α and β are identified on-line using e.g. a recursive least-square algorithm based on previous measurements of L, D, and C_{GC}.

With the heap state prediction of (4.28) and (4.29), (4.31) can be written

$$C_{\mathrm{GC}} \approx (\alpha\, P(Live) + \beta\, P(Dead))\, A \tag{4.32}$$

which, according to (4.30), can be extrapolated:

$$C_{\mathrm{GC}}(k) = (\alpha\, P(Live) + \beta\, P(Dead))\, ((A_s(k) + T_{\mathrm{GC}}(k) \cdot \dot{a}) \tag{4.33}$$

4.3.3 Conservative prediction

The heap state prediction, as presented in Section 4.3.2, depends on $P(Live)$ and $P(Dead)$, in addition to the identified GC work function. For programs with a random, or highly varying, memory usage pattern, the estimates of $P(Live)$ and $P(Dead)$ will contain little information, reducing the quality of the prediction. In such cases, or when robustness is a higher priority than efficiency, a conservative estimate of C_{GC} can be useful. Based on (4.31), it is observed that

$$\alpha\, L + \beta\, D \leq \max(\alpha, \beta)(L + D) \tag{4.34}$$

and $L + D$ is the total amount of allocated memory. Thus, a conservative prediction of C_{GC} is given by extrapolating the amount of allocated memory at the end of the GC cycle, using the amount of memory at the start of the cycle, the GC cycle time, and the allocation rate:

$$C_{\mathrm{GC}}(k) \leq \max(\alpha, \beta)(A_s(k) + T_{\mathrm{GC}}(k) \cdot \dot{a}) \tag{4.35}$$

The main drawback with (4.35) is that the estimate may be very conservative if α and β or L and D are of different magnitude. E.g., if $L = D$, the conservative estimate will be at most twice the true value of (4.31), for any α and β. If, however, the fraction of live memory is only 10%, this method may give an over-estimation of 10 times (in the worst case, $\beta = 0$.) However, for embedded applications it is likely to be reasonable; having very low memory utilization is typically avoided for cost efficiency reasons (and due to the fact that software tends to eventually use all available resources), while a very high memory utilization should be avoided as it causes GC thrashing and poor efficiency [JL96].

4.4 Summary

An approach to making a time-triggered garbage collection scheduler auto-tuning was presented, based on the observation that we need to estimate the two scheduling parameters deadline and execution time. It was shown how a GC cycle time that ensures that the application never runs out of memory can be determined at run-time, and how it is robust against variations in floating garbage. It was also shown how having information about the mutator can be used to reduce the conservativism of the T_{GC} tuning.

Different approaches for on-line estimation of C_{GC} was presented and discussed: first, a black box, "yesterday's weather", approach that

is simple and does not require any information of the state of the memory manager; second, a clear box method based on identifying a GC work function and predicting the state of the heap based on the allocation rate and GC cycle time; and third, a conservative variant of the clear box approach, based on an identified GC work function and worst case assumptions. For the clear box approaches, a simplified representation of the heap state was suggested in order to make implementation practically feasible. Finally, the degree of conservativism in the conservative approach was discussed and it was argued that, for typical embedded systems, it will be within reasonable limits.

On-line estimations of the scheduling parameters for the GC task makes it possible to take the GC overhead into account when doing online schedulability analysis, e.g. in a feedback scheduling system. It also makes it possible to make a semi-concurrent garbage collector adaptive in order to minimize the disturbance of low priority threads. Integrating the scheduling of garbage collection and the scheduling of mutator processes is an important step towards making safe object-oriented languages like Java practically feasible for many real-time applications in automatic control and embedded systems, without requiring a huge engineering effort to tune the GC.

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. Similarly, the presentation focuses on real-time systems, but the proposed mechanisms can be useful in any system where robustness to variations in workload, or isolation between different parts, is required.

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)[2] 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[3] in addition to its core functionality. For example, a digital controller process may produce log data in addition to calculating and outputting its

[2]The terms *critical* and *non-critical* correspond to the terms *mandatory* and *optional* sometimes used in the safety critical systems community.

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

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, for instance by using a bounded buffer for 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 — a mode of operation with lesser performance but radically lower memory consumption that will allow the application to continue executing in an low-on-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 [JP86, SRL94, Per99].

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 semi-concurrent GC scheduling, additional memory allocations in 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.

With priorities for memory allocations, an 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[4], there is no real way of knowing

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

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.

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*[5], 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

[5] 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.

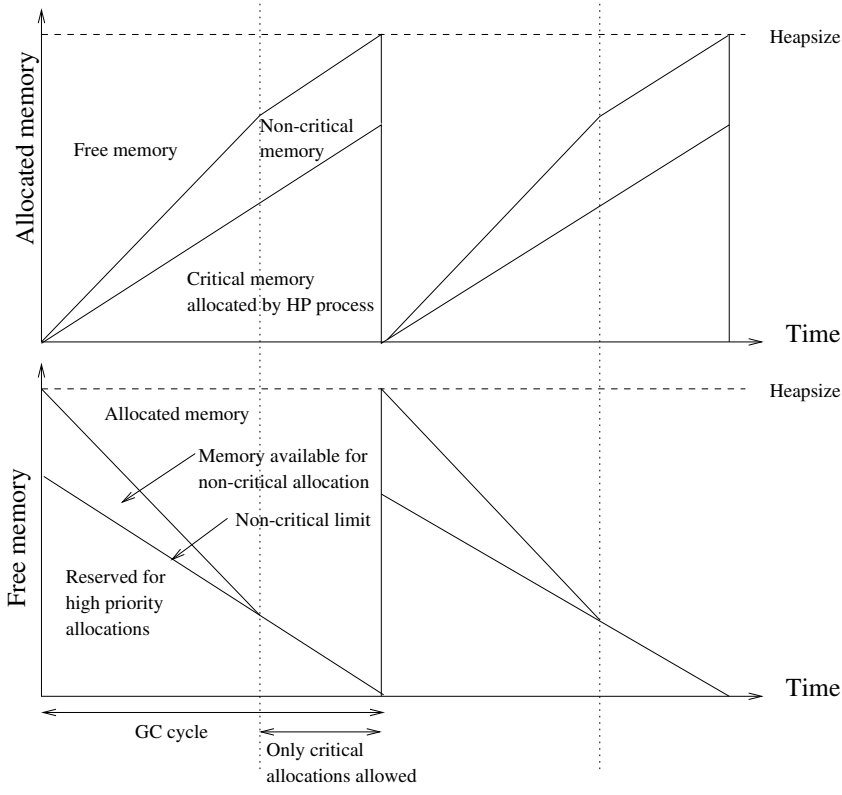


Figure 5.1: *Schematic illustration of the limit for non-critical allocations. The dotted lines indicate the times where the non-critical limit is equal to the amount of allocatable memory, i.e., when the system starts to deny non-critical allocation requests.*

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 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 is 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 in order to allow 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

We will now discuss LP processes in a system with semi-concurrent GC. When LP processes are added 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 is intrinsic to the scheduling principle; 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 critical 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 LP 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 critical memory in the LP processes during this GC cycle, the NC limit can be written

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

Non-critical allocations in LP processes, on the other hand, should not be included in A_{LP}. That means that, if NC LP allocations are made, $L_{NC} > 0$ at the end of the GC cycle, and the total amount of non-critical

allocations allowed during the cycle is not affected by the decrease in cycle time. Just as in the case when there is more available memory than in the worst case, it is not enough to ensure that all HP critical allocations succeed in the current cycle — the ultimate objective is to limit the amount of live non-critical memory.

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[6]. 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 a critical 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

[6]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.

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.

5.4.5 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 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.

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.

Non-criticality is transitive. 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 criti-

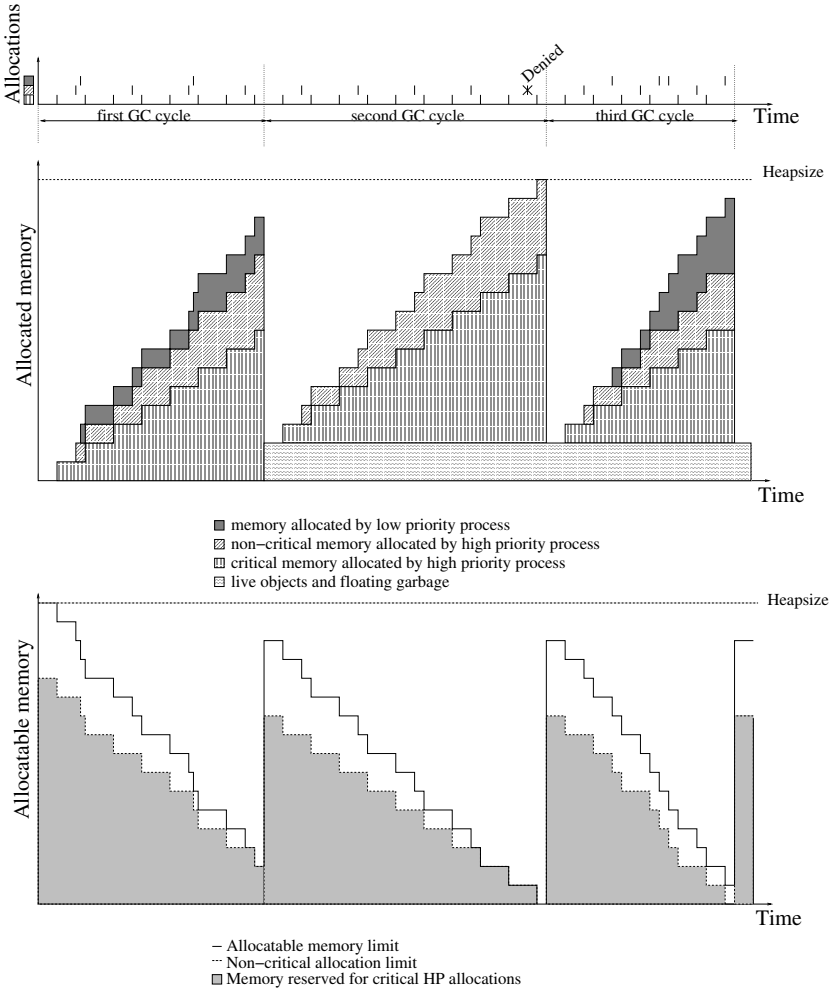


Figure 5.2: *An example showing how the amounts of allocated and allocatable memory vary over time. Allocation requests for non-critical memory are denied when the amount of allocatable memory is less than or equal to the non-critical allocation limit ($R_{HP} - A_{LP}$). This happens at the end of the 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.*

```
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.*

cal 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[7] has been made using the IVM (Infinitesimal Virtual Machine) [Ive03], 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 those 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

It was observed that memory priority and CPU time priority need 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 idea of applying priorities to memory allo-

[7] The experiments with priorities for memory allocations are presented in Section 8.5.

cation 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.

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

CHAPTER 6

MEMORY-AWARE FEEDBACK SCHEDULING

Feedback control is a good way to cope with uncertainties, and has successfully been used in process schedulers for real-time control systems with non-deterministic execution times — a technique known as *feedback scheduling*. Such scheduling is very suitable for systems which change 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, for instance, the period times of the tasks in order to keep the CPU utilization at a safe level while optimizing the quality of service delivered by the system.

This chapter investigates how an auto-tuning time-triggered GC 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 tasks.

It is also studied how the priorities for memory allocations presented in Chapter 5 can be used, in a feedback scheduling system, to control the allocation rates of the application threads in order to optimize the trade-off between memory and CPU time consumption.

6.1 Introduction

Thus far, we have studied how to calculate the scheduling parameters for a time-triggered garbage collector in two different cases. In the first one, all parameters (L_{max}, a_i, etc.) were known and constant. In the second case, the parameters were estimated based on run-time measurements. In a feedback scheduling system, the GC scheduling problem

comes in a third form. Here, the parameters of the mutator threads are known at any particular instant, but may change as the scheduler changes sampling rates in order to maximize the overall performance.

Previous work on feedback scheduling and automatic identification of (soft) real-time systems [AP00] has showed how self-tuning regulators can be used to control resource allocation without a priori knowledge about the task requirements. However, in existing feedback scheduling systems the memory management overhead is either ignored or treated implicitly as a part of the application's execution.

With traditional incremental garbage collectors, the memory management overhead is inlined in the application code, as a small amount of GC work is performed at each allocation. Therefore the memory management overhead can be treated as part of the mutator's execution time and no special consideration is required (although doing so may still improve performance).

With a concurrent garbage collector, that is no longer possible. As the GC work motivated by the actions of the mutator is performed by a separate task, the CPU utilization of that task must be handled explicitly by the scheduler. 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, causing unacceptable delays for the hard real-time tasks. Therefore, 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 calculated by the GC scheduler cannot be changed by the feedback scheduler, but must be taken into account when the period times of the application threads are calculated. This corresponds to making the GC a rigid task in [BLA02].

This chapter studies how to take the memory management costs into account in the period assignment problem of the feedback scheduler. Section 6.2 derives approximative models for estimating and optimizing the GC scheduling parameters together with the period assignment. Section 6.3 briefly discusses how slack in the schedule caused by conservative estimates of the GC utilization can be utilized when the GC finishes its work before its deadline. Section 6.4 investigates how the mechanisms for different priorities on memory allocation requests from Chapter 5 can be utilized in a feedback scheduling context, where controlling the allocation rate of processes gives another degree of freedom when optimizing overall performance.

6.2 GC-aware period assignment

Recall the period assignment problem from Equation (2.3),

$$\min_{h1 \ldots h_n} \sum_{i=1}^{n} J_i(h_i)$$
$$\text{subject to} \sum_{i=1}^{n} \frac{C_i}{h_i} \leq U_{sp} \, .$$

If the cost function, J, is (approximated by) a linear or quadratic function, it has been shown that a closed-form solution to the optimization problem (2.3) can be found: With the cost function

$$J_i(h_i) = \alpha_i + \gamma_i h_i \tag{6.1}$$

or, equivalently,

$$V_i(f_i) = \alpha_i + \frac{\gamma_i}{f_i} \tag{6.2}$$

the optimal frequencies, $f_i^\star$, are given by

$$f_i^\star = \left(\frac{\gamma_i}{C_i} \right) \frac{U_{sp}}{\sum_{j=1}^{n} (C_j \gamma_j)^{\frac{1}{2}}} \tag{6.3}$$

and for a quadratic approximation of the cost function, a similar explicit solution can be found [CEBÅ02].

In a system with a scheduled garbage collector, the required CPU utilization of the GC, U_{GC}, must be taken into account when assigning task periods in order to keep utilization below the setpoint. To get the total CPU utilization U_{sp}, the reference utilization *for the mutator tasks* in the feedback scheduler must therefore be reduced to

$$U_{ref} = U_{sp} - U_{\mathrm{GC}} \, . \tag{6.4}$$

The utilization of the garbage collector is $U_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}}{T_{\mathrm{GC}}}$ and thus, the constraint of the period assignment problem becomes

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{C_{\mathrm{GC}}}{T_{\mathrm{GC}}} \leq U_{sp} \, . \tag{6.5}$$

Given the previously derived expressions for the GC cycle and execution time, we get the the general expression for the required CPU utilization for GC,

$$U_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}(S_h)}{T_{\mathrm{GC}}(H, L, a_1, \ldots, a_n, h_1, \ldots, h_n)}. \tag{6.6}$$

However, at run-time, all parameters are typically not known, and therefore an approximate model must be used. We will now formulate such models for compensating for U_{GC} in FBS period assignment. In the first one, we will simply use a GC auto-tuner, as described in sections 4.2 and 4.3, as a reference generator to the feedback scheduler. In the second, we will incorporate the GC tuning into the optimization problem of the feedback scheduler, in the case where L_{max} is known. In the third one, we assume that L_{max} is unknown and derive similar expressions based on the previously described GC auto-tuning techniques.

As far as the optimization problem is concerned, we will assume that C_{GC} is constant. This just means that in the formulation of the optimization problem, we assume that C_{GC} is independent of the period times of mutator tasks, and that the effects that changes to the schedule has on C_{GC} is captured by the feedback loop. The interaction between the GC cycle parameter estimation and the feedback scheduling is done only through the model for T_{GC}.

6.2.1 Separate GC tuning and feedback scheduler

The most simple way of taking garbage collection work into account is to use the GC auto-tuner as a reference generator for the feedback scheduler. Figure 6.1 shows how the adaptive garbage collection scheduler from Chapter 4 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 deadlines of the application threads 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. In this case, the GC tuner and the feedback scheduler are independent of each other, and the feedback scheduler simply uses U_{ref} as in (6.4), where T_{GC} and C_{GC} are estimated using some of the described techniques.

However, in general, the different tasks have different memory requirements, and thus any changes to the scheduling will affect the GC workload. As the GC scheduler is decoupled from the feedback scheduler, such effects cannot be taken into account in the period assignment, and this is a limitation of the described approach. Instead, any changes to the allocation rate — and, hence, to T_{GC} and U_{GC} — caused by the changes in period times are compensated for by the feedback to the GC tuner. That may, in turn, cause U_{ref} to change, and therefore, this

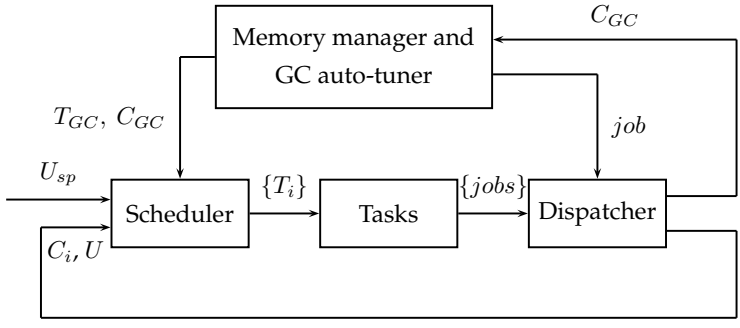


Figure 6.1: *Feedback scheduling of both application tasks and GC. The GC task issues jobs which are dispatched just as any other jobs. The only difference between the GC task and the application tasks is that the GC is allowed to set its own period time while the feedback scheduler changes the application tasks' period times in order to keep $U \leq U_{sp}$.*

model may show oscillating behaviour. Such oscillations can, however, be avoided by using conservative settings in the GC auto-tuner. For instance, if the U_{GC} prediction is filtered using the maximum value and a forgetting factor close to unity, a well damped system can be achieved, at the price of lower average utilization.

Another, and potentially more important, drawback of the separated approach is that the measured GC overhead is divided evenly across all mutator tasks. Thus, even if one task is responsible for the majority of the memory usage, the sampling rates of all tasks will be affected. In systems with competing (as opposed to cooperating) tasks, that may be an issue, as far as fairness in the scheduling is concerned.

6.2.2 Integrated GC and feedback scheduling

If the GC estimation and tuning is incorporated in the feedback scheduler itself, the effects on the GC utilization of changing period times can be taken into account in the period time optimization. In principle, we want to be able to express the cost of garbage collection per task and sample, in a way that the constraint in the optimization problem is on a form that allows the existing closed-form solution to be used.

Under the previously stated assumption that C_{GC} is constant, U_{GC} will be a function of the GC cycle time, which, in turn, depends on the

allocation rate. Thus, we get a utilization constraint with one term for the CPU requirement and one for the memory requirement of each task,

$$\sum_{i=1}^{n} \frac{C_i + K_{\mathrm{GC}} \cdot a_i}{h_i} \leq U_{sp} \tag{6.7}$$

where K_{GC} can be viewed as the cost, in CPU utilization, of memory allocation in CPU seconds per byte. With this formulation, the utilization constraint is of the same form as (2.3), as the extra term is constant (assuming a_i is independent of h_i), and thus the existing explicit solution to the optimization problem can be used. We will now see how the utilization constraint can be expressed when the maximum amount of live memory is known and unknown, respectively.

Using worst case live memory information

Given the maximum amount of live memory, L_{max}, and the amount of memory allocated per period of each task, a_i, we can use Theorem 1 to find the maximum allowed T_{GC} and, hence, the CPU utilization:

$$U_{\mathrm{GC}} = C_{\mathrm{GC}} \cdot \frac{\sum_{i=1}^{n} \frac{a_i}{h_i}}{\frac{H - L_{max}}{2} - \sum_{j=1}^{n} a_j} \,. \tag{6.8}$$

Inserting this expression for U_{GC} into (6.5) gives the constraint

$$\sum_{i=1}^{n} \frac{C_i + \frac{C_{\mathrm{GC}}}{\frac{H - L_{max}}{2} - \sum_{j=1}^{n} a_j} \cdot a_i}{h_i} \leq U_{sp} \tag{6.9}$$

which, assuming that C_{GC} and $\{a_1 \ldots a_n\}$ are independent of $\{h_1 \ldots h_n\}$, can be written as (6.7).

In practice, the period time of the GC will be much longer than that of the mutator tasks, and thus $\sum_{j=1}^{n} a_j$ is typically very small compared to $H - L_{max}$. Further, if a conservative estimation of U_{GC} is used, and $U_{sp} < 1$, there will always be some slack in the schedule. For these reasons, sufficient safety margins can be achieved, making it reasonably safe to approximate (6.9) with

$$\sum_{i=1}^{n} \frac{C_i + \frac{C_{\mathrm{GC}}}{\frac{H - L_{max}}{2}} \cdot a_i}{h_i} \leq U_{sp} \,. \tag{6.10}$$

I.e.,

$$K_{\mathrm{GC}} = \frac{C_{\mathrm{GC}}}{\frac{H - L_{max}}{2}} \tag{6.11}$$

which is precisely the GC CPU time per allocated byte.

Without a priori analysis

The above discussion assumes L_{max} to be known and that it is reasonable to use the worst case live memory. If that is not the case, T_{GC} can be estimated using (4.3), and the constraint (6.5) becomes

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{2\,C_{\mathrm{GC}}}{\frac{F(t)}{\sum_{i=1}^{n} \dot{a}_i} + t - t_s} \leq U_{sp} \tag{6.12}$$

which, with $\dot{a}_i = \frac{a_i}{h_i}$, gives

$$\sum_{i=1}^{n} \frac{C_i}{h_i} + \frac{2\,C_{\mathrm{GC}}}{\frac{F(t)}{\sum_{i=1}^{n} \frac{a_i}{h_i}} + t - t_s} \leq U_{sp} \tag{6.13}$$

which can be reorganized as

$$\sum_{i=1}^{n} \frac{C_i + \frac{2\,C_{\mathrm{GC}}}{F(t) + (t-t_s)\sum_{i=1}^{n} \frac{a_i}{h_i}}\, a_i}{h_i} \leq U_{sp}\,. \tag{6.14}$$

I.e.,

$$K_{\mathrm{GC}} = \frac{2\,C_{\mathrm{GC}}}{F(t) + (t - t_s)\sum_{i=1}^{n} \frac{a_i}{h_i}} \tag{6.15}$$

Unfortunately, the constraint (6.14) is not linear, meaning that the existing closed-form solution is not directly applicable. Worse yet, in this form, we get an optimization problem where both the objective function and the constraint are concave, and that makes it practically useless.

In order to remedy that, an approximation that turns (6.14) back into a linear constraint is sought. It is observed that, if $\dot{a}$ is constant, the denominator in (6.15) is equal to $F(t_s) = F_s$. If that is used to linearize the constraint, we get

$$\sum_{i=1}^{n} \frac{C_i + \frac{2\,C_{\mathrm{GC}}}{F(t_s)}\, a_i}{h_i} \leq U_{sp} \tag{6.16}$$

and

$$K_{\mathrm{GC}} = \frac{2\,C_{\mathrm{GC}}}{F_s}. \tag{6.17}$$

The error in the T_{GC} approximation of (6.16) will increase with increasing changes in $\dot{a}$ and the effect will be greater if the change occurs later in the GC cycle. Figure 6.2 shows how the approximation error depends

on the change in $\dot{a}$ and the time of change. For instance, if the allocation rate is doubled half-way into the GC cycle, the relative error in the T_{GC} approximation will be $20\,\%$. However, as the total GC utilization typically is 5–$20\,\%$, the overall impact of the error in the approximated utilization will only be a few percent. For robustness, a safety margin to accommodate such uncertainties can be added when setting U_{sp}.

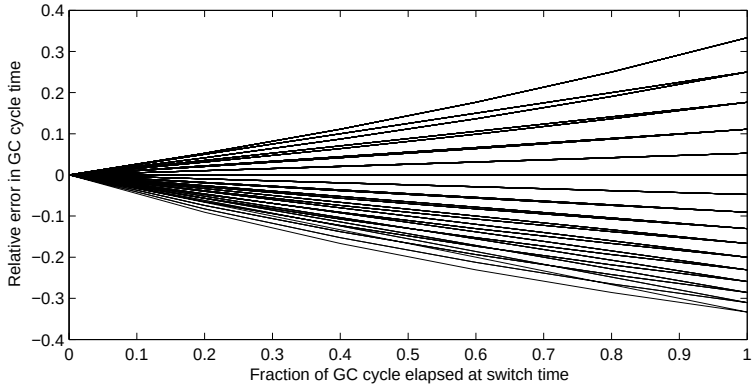


Figure 6.2: *Relative error in T_{GC} approximation as function of change in $\dot{a}$ and time of switch. The lines represent changes in $\dot{a}$ from a factor of 0.5 to a factor of 2. An increase in $\dot{a}$ causes underestimation of U_{GC}.*

Thus, with suitable approximations, the CPU requirement of the GC task can be included in the period assignment, while keeping the optimization problem on a form that allows the existing closed-form solutions to be used.

6.3 Utilizing slack

By making the costs of memory management explicit and taking them into account in the period time optimization, it is possible to use a concurrent garbage collector in a feedback scheduling system. In order to get a system that is robust to variations in execution times, the utilization setpoint is typically set below $100\,\%$. Also, to get stable estimates of GC scheduling parameters, the estimation needs to be conservative. That means that, in the average case, there will be some slack in the schedule, allowing the GC to finish before its deadline.

The feedback scheduler reserves a fraction of the CPU time for garbage collection. However, when the GC is not running, this CPU time could

be used for mutator threads. In a system with a time-triggered GC, it is known that when the GC has finished a cycle it will not need to run again until at its next release time. If the feedback scheduler is aware of the state of the GC, this means that when the GC has completed a cycle, a higher mutator utilization can be allowed until the next GC release time. That is, if the GC finishes at time t_f ; $t_s < t_f < t_e$,

$$U_{ref}(t) = \begin{cases} U_{sp} - U_{\mathrm{GC}}, & t_s \leq t \leq t_f \\ U_{sp}, & t_f < t < t_e - \delta \end{cases} \tag{6.18}$$

where δ is used to take into account the fact that increasing the mutator utilization may increase the allocation rate and, hence, shorten the time until the next GC release.

The GC cycle time, and consequentially, the start time of the next GC cycle, was estimated based on worst case assumptions about floating garbage, but when a GC cycle has finished, it is known how much memory was actually reclaimed. Thus, δ depends on both the amount of free memory and the allocation rate. We know that the amount of free memory at the time the GC has completed the cycle, $F(t_f) \geq F_{min}$. The requirement is the same; when the next GC cycle starts, the amount of free memory must be no less than F_{min}. Therefore, the adjusted release time of the next GC cycle must satisfy

$$R'_{\mathrm{GC}}(\dot{a}) \leq t_f + \frac{F(t_f) - F_{min}}{\dot{a}} \tag{6.19}$$

and, with equality, we get

$$\delta = t_e - R'_{\mathrm{GC}}(\dot{a}) = t_e - \left(t_f + \frac{F(t_f) - F_{min}}{\dot{a}} \right). \tag{6.20}$$

Thus, a sufficient degree of conservatism can be used to give robustness against inaccuracies in the GC scheduling parameter estimates due to variations and approximations, without the low average CPU utilization normally associated with such conservative scheduling.

6.4 Controlling the allocation rate

As we have seen, the fraction of CPU time that must be reserved for garbage collection depends on the allocation rate of the mutator, which, in turn, depends on the period times of the individual threads. Therefore, in a system with garbage collection, the feedback scheduler controls

the CPU usage of a thread directly, through the period assignment, but also indirectly as the period time affects the allocation rate.

The notion of priorities for memory allocations, introduced in Chapter 5, suggests that it may be possible to, to some extent, directly control the allocation rates of the individual threads. Having such a mechanism may be used to increase the flexibility of a feedback scheduler, by making it possible to separate allocation of memory and CPU time. As higher memory usage means more GC work, that allows the scheduler, or resource manager, to trade off memory usage for CPU time.

Assuming that each task has a critical and a non-critical part, with memory requirements of $a^{(c)}$ and $a_{max}^{(nc)}$, respectively, we extend the cost function with a term corresponding to the increase in quality from the non-critical parts

$$J(h, a^{(nc)}, \ldots) = \ldots \; ; \; 0 \leq a^{(nc)} \leq a_{max}^{(nc)} \tag{6.21}$$

which gives the optimization problem

$$\begin{aligned} \min_{h1 \ldots h_n} \quad & \textstyle\sum_{i=1}^{n} J_i(h_i, a_i^{(nc)}, \ldots) \\ \text{subject to} \quad & \textstyle\sum_{i=1}^{n} \frac{C_i + K_{\text{GC}} \cdot \left(a_i^{(c)} + a_i^{(nc)}\right)}{h_i} \leq U_{sp} \end{aligned} \tag{6.22}$$

The motivation for introducing different priorities for memory allocations as presented in Chapter 5 was primarily to provide isolation between critical and non-critical parts of a system. Now, we focus on optimizing the performance of the application, and thus it becomes more important to take *which* allocations that should be preformed into account. The memory manager can, however, only limit the amount of non-critical allocations per time unit (typically, per GC cycle); as the run-time system doesn't have any information about the purpose of the application threads it cannot make any detailed decisions about exactly which allocations to allow or deny.

In order to maximize the quality of service, it is therefore better to actually communicate how much non-critical memory it is currently allowed to use to each thread. In the application, this can then be translated into performing some parts every nth sample, or something similar. Thus, while the hard limit used to ensure robustness may be enforced by the run-time system, the programmer can make more fine-grained decisions about how make best use of the memory available to each thread.

In an actual implementation of these ideas in a feedback scheduled system, the memory non-critical limit must be set individually for each

thread and thus the interface between the feedback scheduler and memory manager must contain operations for that. Or — if the threads are cooperating — it can be expressed directly in the code.

In the general formulation, (6.22) might prove hard to solve on-line. In order to test the fundamental principle, we will now investigate a simplified case, where the problem is reduced to either allowing all or no non-critical allocations of a thread in each sample.

Case study: Ball-and-beam

As an example, we take the ball-and-beam process[1], controlled by a LQG regulator. It is assumed that the angle can either be measured (which requires a measured-value object to be allocated and passed to the controller) or estimated by using an observer. I.e., the allocation of the angle measurement is non-critical. Depending on the state of the memory system, K_{GC} — and hence, the total available CPU utilization — will vary.

Using Matlab-based tools, the effects of the scheduling on control performance in the described scenario is analysed and simulated. For control performance analysis, the Jitterbug toolbox is used. Jitterbug is a tool for studying how timing affects the performance of a computer-controlled system [LC02]. The simulation was done using the TrueTime real-time kernel simulator in Matlab/Simulink, with simulated heap, a separate GC thread, and one disturbance task. The simulator is presented in Chapter 8.

Theoretical analysis

The analysis is done for two versions of the ball-and-beam controller: with or without angle measurements. The controller with the angle measurement will allocate more memory per sample, and therefore, under the discussed feedback scheduling, it will suffer a bigger penalty from the GC overhead. On the other hand, for the same sampling rate, the controller using angle measurements will perform better. In order to optimize quality of control, the cost of memory management must be balanced against the control performance, to choose which of the two controllers to use, given a certain K_{GC}.

Figure 6.3 shows the calculated total cost for a range of sampling rates. Figure 6.4 shows the sampling rate for the two controllers as a function of K_{GC}. The controller without angle measurements has lower memory requirement, and is therefore much less sensitive to K_{GC}.

[1] The experiment setup is described in more detail in Chapter 8.

Putting this together, using linear cost functions, Figure 6.5 shows J as function of K_{GC} for the two systems. The intersection of the lines is the value of K_{GC} where the system with observed angle starts outperforming the one with measured angle as a lower memory usage allows a higher sampling rate — the optimal K_{switch}.

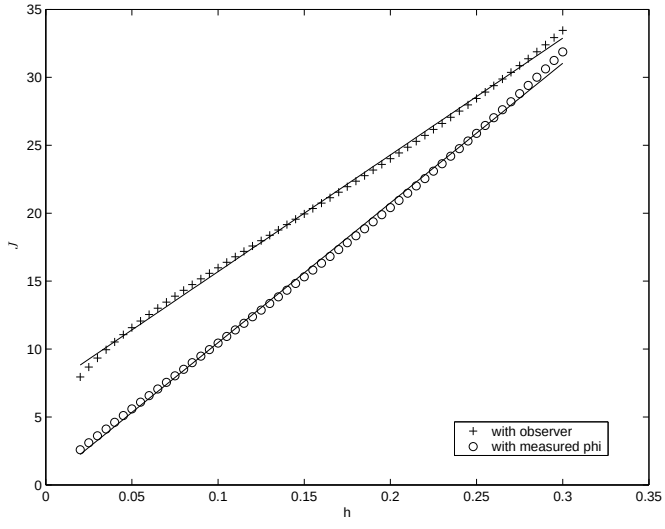


Figure 6.3: *The calculated costs and the linear approximations.*

Simulation

In order to measure the control performance of the LQG regulator, if q_n is the weight of the nth state (i.e., $Q = \mathrm{diag}(q_1 \dots q_n)$), and x is the state vector, we define the total cost as

$$J_{tot} = \int^{t} \sum_{i=1}^{n} q_i x_i^2(t)\, dt\ . \tag{6.23}$$

Running the system with different values of the switching point K_{switch} and measuring J_{tot} gives the plot shown in Figure 6.6, where the minimum corresponds to the optimal K_{switch}. The cost is the total cost of a $160s$ execution, and it is not normalized. The absolute values of the cost are not very interesting, as a direct comparison with the analysis is not possible as they show different things. The analysis calculated the cost for different, constant, values of K_{GC}. In the simulation, K_{GC} varied

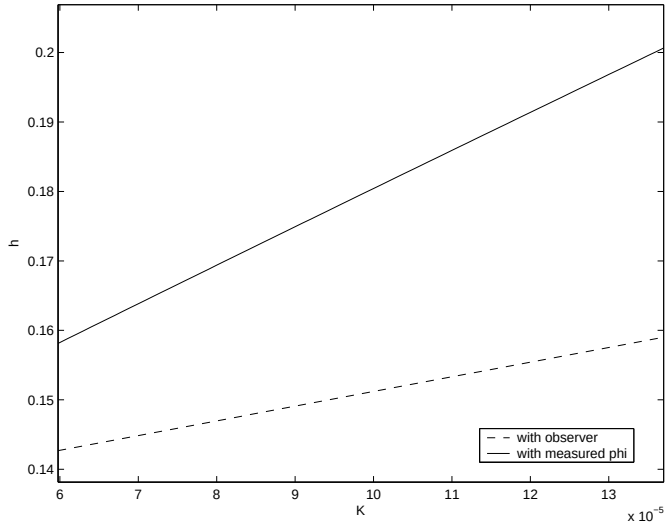


Figure 6.4: *Sample rate as function of K_{GC}.*

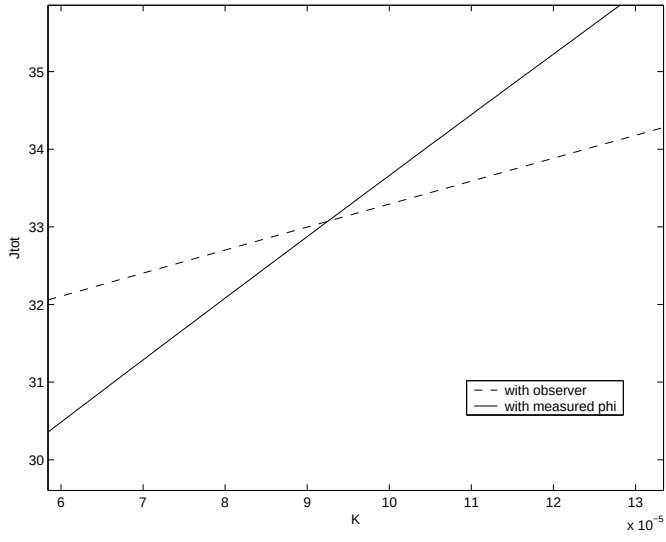


Figure 6.5: *Cost as function of K_{GC}. For high values of K_{GC}, the system with the observer will outperform the one with measured angle, due to the large CPU cost of memory allocation.*

throughout the execution and at each scheduling instant the controller with the lowest cost was used. The GC was scheduled as described in Chapter 4, and the feedback scheduler used U_{GC} to adjust the utilization reference, according to Section 6.2.1.

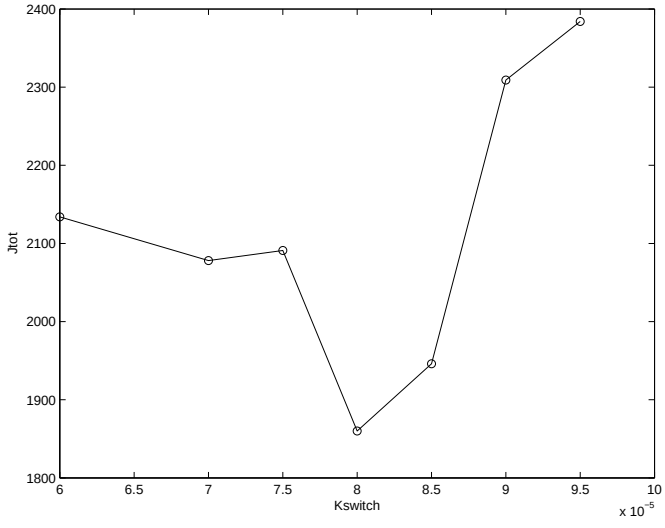


Figure 6.6: *Total cost as function of K_{switch}.*

In theory, the minimum in Figure 6.6 should be at the same K_{GC} value as the intersection of the lines in Figure 6.5. The discrepancy between the theoretical and simulated results can be explained by a combination inaccuracies in both the models and the run-time system. The theoretical results are based on an optimal feedback scheduler, but at runtime, some approximations are required. Notably, in order to determine the mutator utilization, both the GC cycle time and the GC execution time has to be predicted. The GC cycle time is dependent on the allocation rate and object distribution, which are both affected by the mode changes. Also, in order to get a high enough K_{GC} to reach K_{switch}, the system had to be quite stressed, with a U_{GC} around 45 – $55\,\%$. Thus the impact of the discussed approximations and uncertainties, which would be small in a system with lower U_{GC}, became significant.

While the setup in this simple case study is not entirely realistic, it still illustrates the fundamental idea that if memory usage can be controlled, the total quality of control of a system can be improved by online optimization of the trade-off between memory and CPU usage.

6.5 Summary

In order to use scheduled garbage collection in a feedback scheduling system, the required CPU utilization of the GC task must be known, and as the GC utilization depends on the memory behaviour of mutator tasks, it must be determined on-line. Also, as feedback scheduling is typically used in systems where the workload of the mutator tasks (and, hence, their execution pattern) is variable, the GC scheduling cannot be static but must be able to react to such changes.

Different approaches to taking the GC into account in the period assignment of a feedback scheduler were suggested. In the first approach, the GC auto tuner is used as a reference generator to the feedback scheduler, using feedback to adjust the utilization reference based on measured and estimated GC utilization. In the second approach, the GC scheduling is incorporated into the period assignment of the feedback scheduler.

Both approaches have similar performance[2], and the major differences between them lie in implementation and fairness. The separate approach is easier to implement, as the communication between the feedback scheduler and the memory manager is kept at a minimum: the GC utilization is accounted for by changing the utilization reference of the feedback scheduler. The advantage of the integrated approach is that it increases fairness of the schedule, as the memory usage is accounted for as a part of the execution time of a task.

Feedback scheduling is a technique for on-line resource management. It was suggested that overall performance can be enhanced if also memory usage could be included in the optimization. If a controller can be run in different modes, with different memory requirements, the trade-off between memory usage and CPU usage can be optimized on-line.

This chapter has presented different examples of how communication between the memory manager and feedback scheduler is, to some extent, necessary, and, in other cases, opens new possibilities for optimization of the performance of the complete system.

[2]Experiments are presented in Section 8.6

CHAPTER 7

GC IN AN UNCOOPERATIVE ENVIRONMENT

Due to external requirements, run-time systems for embedded applications may have to operate in an uncooperative environment; for instance, extra-functional requirements or historical reasons may stipulate using an off-the-shelf C compiler and RTOS or including external, legacy or automatically generated, C code. In such cases, one cannot rely on detailed assumptions on the behavior of the back-end C compiler or the thread scheduler, which makes implementation of a real-time GC more challenging. For instance, it means that any synchronization required between collector and mutator, needs to be done explicitly. It also means that the generated C code must be written so that it ensures, in a portable way, that no back-end optimization causes interference with the GC.

In particular, the combination of uncooperative compiler, uncooperative scheduler, and tight real-time requirements (low latency) makes a demanding challenge. Without control over the scheduling, some compiler optimizations cannot be allowed, as threads may be preempted at any time. For instance, if we are using a copying or compacting GC algorithm, pointers must always be read from memory, and not kept in registers, as the collector may move objects at (from the mutator's point of view) any time. Furthermore, explicit synchronization with the collector is required, which adds to the execution time overhead of memory operations.

It is shown, and experimentally verified, how it is possible to implement an accurate, concurrent GC in an uncooperative environment, with maximum latency times of a few microseconds, and acceptable run-time overhead. Potential bottlenecks are identified, and compile-time and run-time optimizations to mitigate the problems are suggested.

After the introduction in Section 7.1, Section 7.2 discusses the problems associated with concurrent GC in an uncooperative environment and presents our approach. Section 7.3 briefly describes our GC API. Section 7.4 investigates some potentially expensive performance bottlenecks and Section 7.5 discusses how they may be mitigated.

7.1 Introduction

In a run-time system for real-time Java, or other safe languages with automatic memory management, it is essential to have an accurate, or exact, (i.e., non-conservative), concurrent garbage collector with adequate real-time performance. Due to the external requirements discussed below, the GC must also be able to function in an uncooperative environment, meaning that we must make sure that neither correct behaviour, nor real-time performance, is jeopardized by compiler optimizations, concurrency issues or interference from external code. The challenges encountered when designing and implementing such a run-time system include:

Real-time performance The collector should be fully concurrent in order to make it possible to schedule GC in a non-intrusive way [Hen98, RH03]. It should also have very fine-grained incrementality to allow latency times of at most a few microseconds as required e.g. in automatic control applications.

Usability and flexibility Just as a system must make efficient use of system resources, it must not require unreasonable amounts of engineering effort in order to meet e.g. timing and space constraints. Therefore, an important requirement on a run-time system that is to be practically usable is that it is easy to use and offers sufficient flexibility. This means, for example, that the interface to the memory manager must be fairly simple, and that it should be flexible enough to allow migration between platforms, operating systems, and GC algorithms with little, or no, effort.

Uncooperative compiler The major reason for using a non GC aware compiler is availability; there are C compilers for practically all computer platforms and therefore it is desirable to implement new compilers for high level languages using C as intermediate code and a standard C compiler as the back-end. This gives access to a portable and highly optimized back-end without having to spend the effort required to implement one. However, not having control

over machine code generation makes it harder to implement accurate garbage collection. Finding roots and identifying references are more difficult as we do not have control over activation record layout, register allocation, etc.

Uncooperative scheduler As with compilers, there are many reasons for wanting to use an off-the-shelf real-time operating system. Also, if there is a need to call external native code it is not possible to rely on specific scheduling features like preemption points to ensure safe behaviour, as such external code does not contain preemption points. For instance, a thread in a Java program may call native functions in legacy libraries or code generated from a tool (like Matlab/Simulink). As it must be possible to preempt a thread during such native calls, if we want to make guarantees on latency, preemption must be possible at any instant and not just at preemption points.

Furthermore, for the sake of portability (currently, our system runs on posix[1], Linux/RTAI[2], STORK[AB91], and a locally developed kernel for the Atmel AVR series of micro-controllers) the interface to — and reliance on certain features in — the underlying OS must be kept at a minimum.

In isolation, each of these aspects do not pose a large problem, but, as we will see, the difficulty comes from the combination, which generates conflicting requirements. In particular, synchronization between the mutator and collector can be a problem; full preemption in combination with a fully concurrent GC — especially a compacting one — requires mutual exclusion and, to get short latency times, quite frequent locking and unlocking, which may be a serious performance bottleneck.

Since we usually cannot expect to have control over every aspect of the execution environment of an embedded control system, we need to find a way to handle the conflicting requirements this places on the design of a concurrent GC. This includes finding out how to make a reasonable trade-off between short latency and overall performance as well as investigating what possibilities exist for reducing the impact of the design conflicts by applying a combination of compile-time and run-time optimizations.

[1] Tested on Solaris on UltraSparc and Linux on Intel and PowerPC

[2] The DIAPM Real-Time Application Interface is an addition to Linux, making it possible to run hard real-time tasks at the kernel level, below Linux. Tested on Intel, PowerPC and Axis ETRAX computers.

7.2 Exact GC in an uncooperative environment

The desire to use standard real-time operating systems and standard C compilers means that we will have to construct a GC that not only is independent of operating system or compiler support, but will work *despite* the way the operating system and compiler works. Designing a concurrent GC for such an environment is a challenge. The GC must be synchronized with the application threads and the operating system such that the heap remains consistent no matter how the operating system chooses to schedule the system. The compiler must also be prevented from certain optimizations that could jeopardize the integrity of the heap — especially in combination with a preemptive scheduler. We must also provide some type of runtime type information in order to make it possible for the GC to identify all references.

7.2.1 Uncooperative compiler

We must ensure that all references are traversed by the collector. This includes finding references that reside in local variables as well as avoiding that references are missed because of compiler optimizations. References may reside on stacks or on the heap (or in registers, but this must be avoided). If the back-end compiler doesn't know about references, the code has to contain explicit instructions to inform the GC about the location and scope of each reference.

A common way of implementing this is by pushing the location of any local reference variable onto a *root stack* [Hen98]. Another alternative is to group all local variables for each function together into a C struct and to link these structs together forming a *shadow stack* containing all references [Hen02].

Finding references on stacks

An accurate traversing GC must be able to correctly identify all references outside the GC heap which reference objects on the heap. I.e. it must find all the *root references*. Roots can consist of global variables as well as variables located within method activation records on the C stack. An efficient strategy for tracking these is required.

Keeping track of root references becomes especially hard when we use a compiler or a back-end without support for GC, since we have little or no control over activation record layout, register allocation, and code optimization. In order to gain independence from the compiler, we need to have a known — to the GC — format of references. Just storing

references as C pointers is not possible, as any kind of optimization may then be performed on them.

Our approach to tracking root references is to use an auxiliary root stack, consisting of reference structures, as shown in Figure 7.1. The reference structs also reside on the C stack and are linked together in a singly linked list. Roots are registered by calling `PUSH_ROOT(gc_root)` and de-registered by calling `POP_ROOT(gc_root)`[3].

```
typedef struct gc_root {
  GC___REF(ObjectHead) ref;
  struct gc_root *next_root;
} gc_root;
```

Figure 7.1: *The structure used to track references on stacks. The `GC__REF(type)` macro expands to `type*` for a non-moving GC, or to `type**` for a moving GC (to accommodate the indirect table or forwarding pointer of the read barrier).*

In a multi-threaded system, each thread has its own stack, and we use one root stack per thread. This makes things like popping all local variables of a function and handling exceptions easier, and also reduces the amount of required synchronization as the thread root stacks are independent. The roots of each thread are kept in a linked list (as above), and the heads of each thread root stack (marked, in our implementation, by `next_root==0`) are kept in a doubly linked list. The structure is shown in Figure 7.2, and Figure 7.3 gives an example of how the set of root stacks are linked together.

Finding references in objects

In order to find references in objects (i.e., on the heap), we need to have information about the object layout. This can be implemented in several ways. One method is to associate a *trace function* with each object type which calls a GC function for each reference in the object. Another alternative is to insert into each object a reference to another object that contains information about the layout of the object. We call this information *GC info*. The GC parses this information in order to find the references to traverse.

[3]In our implementation, we use the root structure as a stack. As roots are typically local variables, their lifetimes depend on the scope they are declared in, and thus, exhibit a stack-like behaviour. However, as the roots are kept in a list, it is possible to register and de-register roots in an arbitrary order.

```
typedef struct gc_root {
  GC___REF(ObjectHead) ref;
  struct gc_root *next_root;
  struct gc_root *top_root;
  struct gc_root *next; // next thread
  struct gc_root *prev; // previous thread
} gc_root;
```

Figure 7.2: *The root layout for multi-threaded programs. The* `top_root`*,* `next` *and* `prev` *fields are not used for the actual root elements, so this memory doesn't need to be allocated for the roots, but having the same struct for both list heads and list nodes simplifies the traversal code.*

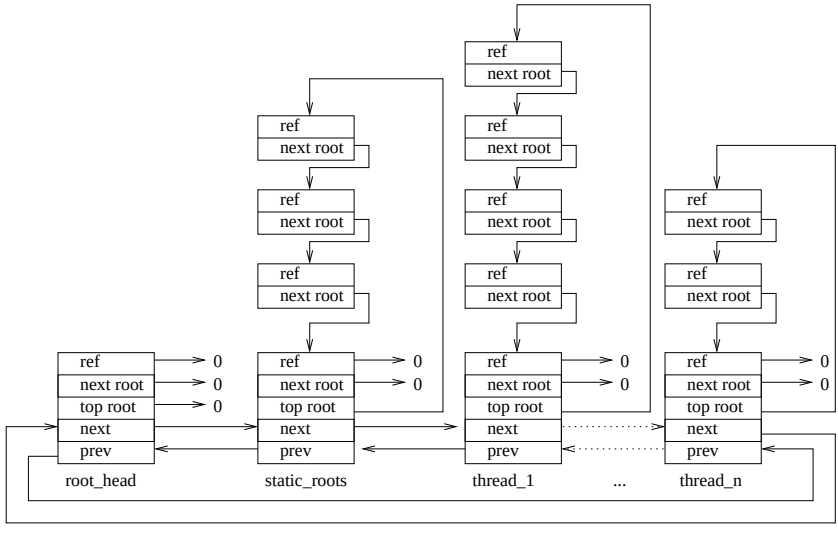


Figure 7.3: *Data structure for roots. One root stack per thread and one for static (system) objects. For the thread root stack heads,* `ref == null`*.*

Each object in our implementation contains a reference to a template object. The template object contains various runtime type information about the object including the GC info and object size. The layout of the GC info is a zero-terminated array $[R_0, D_0, R_1, D_1, \ldots, R_N, D_N, 0]$, where R is the number of references and D the number of data bytes, as is shown in Figure 7.4. In addition, we use special escape codes to indicate that the size of a variable size array of either references or data is stored in the object itself. This makes it possible to use the same GC info object for arrays that only differ in length.

By using an object layout convention which says that all objects should start with a sequence of all references followed by all data fields of the object, the total size of the GC info can be reduced to three integers. This can be generalized to that the size of the GC info array is limited to $2N+1$ where N is the depth of the inheritance hierarchy of that object.

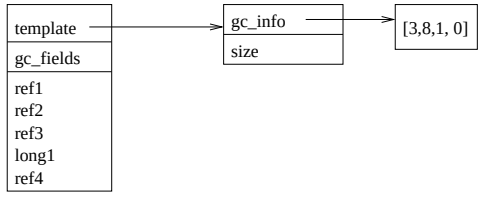


Figure 7.4: *Object layout example: The object consists of three references followed by eight bytes of data, and finally one more reference.*

Ensuring safety

A problem for garbage collectors in uncooperative environments is that compiler optimizations may make it harder to find roots. For instance, if a reference variable is allocated to a register and never stored on the stack, it will not be found by the garbage collector. A conservative collector that relies on heuristics and assumptions on the stack frame layout is vulnerable to this type of problems, but this is not the case in the presented approach.

As our GC uses its own auxiliary structures to find roots on the stack, the only requirement is that all reference variables are stored in memory when the GC runs. This can be expressed in standard C by taking the address of any local reference variable (the `&var` construction). Then, the compiler must allocate that variable on the stack (as it must have a memory address) and ensure that it is written back to memory before

function calls. As we have explicit instructions for linking local roots into our root structure, this requirement is fulfilled. Depending on how the read barrier is constructed, reference variables may also need to be declared `volatile` to ensure that they are read from memory, as the GC thread may have moved objects.

7.2.2 Uncooperative scheduler

With a fully concurrent GC, and without control over the thread scheduler, we must ensure that a context switch does not cause a process to be left in an inconsistent state. For example, we must prevent that a thread is preempted in the middle of the execution of a read- or write barrier. A context switch may occur at any time, which means that the mutator and collector must have mutually exclusive access to the heap in order to prevent both that a process is preempted during reference operations and that a reference or an object is read when the heap is in an inconsistent state due to GC.

It should be noted that the need for synchronization between a concurrent GC and mutator threads described here is not a result of (ahead-of-time) compilation; the same issues arise in e.g., a JVM using native threads. An uncooperative compiler complicates things further as it imposes restrictions on the implementation, but the fundamental problem is that certain memory accesses and reference operations must be atomic to the mutator and collector even if they are not atomic from scheduler's point of view.

Locking the heap can be accomplished in various ways depending on the current platform and operating system. A method that might be advantageous on some platforms is to disable interrupts when exclusive access is required in order to prevent context switches. In many situations disabling interrupts is not feasible; for instance the operating system might prevent programs from doing so, or it might interfere with other parts of the system. In such cases, the synchronization mechanisms provided by the operating system must be used, e.g. a semaphore or a monitor.

7.3 Garbage collector interface

Different GC algorithms require different interaction with the application. For instance, a compacting or copying collector requires a read-barrier, as objects may move, where a non-moving mark-sweep collector only requires a write barrier. These differences makes it error-prone

and troublesome to write code generators supporting more than just one type of GC algorithm, and it gets even worse considering hand-written code, which would need a major rewrite for each supported GC type.

In order to separate, and hide, the GC implementation from the application, we have specified a garbage collector interface (GCI) that provides heap access primitives to the application [IBE+02]. This includes providing the necessary synchronization for reference and heap operations. The GCI is used both in the Infinitesimal Virtual Machine and in the LJRT compiler, and with both concurrent and stop-the world versions of non-moving, compacting and copying collectors. By using the interface, no changes to the compiler or VM is required, when a new GC is added.

As the efficiency of a garbage collection algorithm is highly dependent on the behaviour of the application, the choice of GC algorithm is part of the configuration and tuning of a system. The separation provided by the GCI means that the intermediate C code generated by the LJRT compiler doesn't have to be re-generated in order to change GC, only C compilation is required, which makes experimenting with different GCs quicker and easier.

The varying requirements of different GCs, both on the set of memory access primitives and run-time aspects causes the interface to contain quite many operations, which makes it less than ideal for manually written code, but as the main intended use for GCI is generated code (especially from our Java to C translator) or low-level routines in a virtual machine, this is no major concern. As always, there is a trade-off between keeping the interface small and limiting the power of expression as little as possible. Manually writing code that accesses the heap through the GCI is, however, also quite doable.

The interface consists of primitives for initialization, object layout declaration, reference variable declaration, object allocation, reference access, field access, and function declaration and call, which adds up to 50 primitives. The GCI is implemented as C macros, and, as an example of how the interface looks, we take the field reference operation: reading a reference field from an object is done through the `GC_GET_REF` macro. Figure 7.5 show how an expression of the type `t = a.b.c`, on an object `a`, must be split up to fit the GCI. Note that a temporary variable (`tmp`) is used and how it is pushed onto and popped from the root stack. The first `GC_GET_REF` macro expands to executing the read barrier on `a` (i.e., finding a pointer to the actual object), assigning `a.b` to `tmp` and executing the write barrier.

```
GC_REF(Type, tmp);        // Type tmp;
GC_PUSH_ROOT(tmp);
GC_GET_REF(tmp, a, b);    // tmp = a.b;
GC_GET_REF(t, tmp, c);    // t = tmp.c;
GC_POP_ROOT(tmp);
```

Figure 7.5: *Example of field access through the GCI.*

7.4 Performance issues

This section discusses the run-time overhead incurred by the presented approach and Section 7.5 presents some optimizations that reduce that overhead. The problems described here are to a large part due to the fact that we have conflicting requirements on our design. As stated earlier, the design criteria behind our system is that it should

- cause very low latency
- not require compiler (back end) cooperation
- not require scheduler cooperation
- have low execution time overhead

A combination of any three is fairly easy. The problem is achieving all these properties at the same time. If we add the requirement that the implementation should have a simple interface and high flexibility, it gets even more difficult.

As we cannot rely on cooperation from the scheduler, the application[4] code must provide the required synchronization. In order to keep the latency low, tight synchronization with very small critical sections is required, and our experiments show that the dominating part of the overhead is introduced by frequent locking, so the discussion will focus on that. Furthermore, the requirement on simplicity and flexibility must also be taken into account, which means that we cannot rely on manual tweaking in order to meet the real-time requirements. However, even if a very large engineering effort can be put into manual tuning, there will still be an overhead due to the synchronization, compared to a traditional non-real-time GC.

It is stressed that the discussion in this chapter is based on the above assumptions, and that the performance problems described are caused by the high degree of synchronization necessary when we require very

[4]From the operating system's point of view; *application* includes the GC.

low latency and are constrained by an uncooperative environment (in particular the scheduler) and/or unknown external native code. If, on the other hand, the scheduler and application allows it, preemption points could be used; if preemption points are placed in a way that ensures that preemption only may occur when the heap is in a consistent state, no additional synchronization would be required. This is a commonly used technique, but as it is not suitable for our applications, it is outside the scope of this discussion.

7.4.1 Too frequent locking

As stated, in order to achieve short latency times, we need to make the atomic GC operations as short as possible. As the heap must be locked during GC operation (to provide mutual exclusion w. r. t. the mutator) this requires frequent locking and unlocking. Even though each lock-/unlock operation is really cheap, the number of lock/unlock operations in the straight-forward implementation proved to be a serious performance bottleneck. For the test application presented in the experiments, the straight-forward implementation performed thousands of lock/unlock operations per sample. This was the major limiting factor on the possible sample rate for the controller.

The assignment statement `t = a.b.c` of the example in Figure 7.5, illustrates the problem with locks: if we expand the GC macros to show the locking instructions, it looks like in Figure 7.6. In this example, there are three pairs of `gc_unlock(); gc_lock();` instructions, due to the fact that many atomic operations are executed in sequence, and that each operation has to contain the proper synchronization. This is obviously quite inefficient. If the lock and unlock instructions could be placed arbitrarily, the intermediate pairs could be removed in order to to increase efficiency, resulting in the code as shown in Figure 7.7, reducing the locking overhead by 75%. Also, as no GC work may occur while the heap is locked, the `GC_PUSH_ROOT()` and `GC_POP_ROOT()` operations may also be removed, leaving us with just the code in Figure 7.8, which is much more efficient and has almost as small critical section as each of the original primitives. This is, of course, a very simple example. In a real program the lock instructions could be placed at arbitrary intervals, which allows the trade-off between latency and throughput. Nonetheless, the example illustrates the problem of big synchronization overhead due to small atomic operations.

The optimization described in the above example, however, requires both that the locking instructions are accessible in the interface and either tedious manual placement of locking instructions or that we have

tool support (in the compiler and/or in the run-time system) for automatically inserting lock/unlock instructions at suitable (for a certain desired latency) intervals.

```
GC_REF(Type, tmp);
gc_lock(); GC__IMPL_PUSH_ROOT(tmp); gc_unlock();
gc_lock(); GC__IMPL_GET_REF(tmp, a, b); gc_unlock();
gc_lock(); GC__IMPL_GET_REF(t, tmp, c); gc_unlock();
gc_lock(); GC__IMPL_POP_ROOT(tmp); gc_unlock();
```

Figure 7.6: *Example showing the expanded macros, revealing the lock instructions enclosing the implementation-layer macros.*

```
GC_REF(Type, tmp);
gc_lock();
GC__IMPL_PUSH_ROOT(tmp);
GC__IMPL_GET_REF(tmp, a, b);
GC__IMPL_GET_REF(t, tmp, c);
GC__IMPL_POP_ROOT(tmp);
gc_unlock();
```

Figure 7.7: *Example with bigger critical section*

```
GC_REF(Type, tmp);
gc_lock();
GC__IMPL_GET_REF(tmp, a, b);
GC__IMPL_GET_REF(t, tmp, c);
gc_unlock();
```

Figure 7.8: *Example with root operations removed*

The heap synchronization is not trivial when really fine-grained incrementality is desired, and having explicit locking instructions increases the risk of concurrency errors, as the responsibility would be moved from the GC to the application code. It would also contradict the design goal of GCI, that the details of the different GC implementations (in this case, the synchronization) should be hidden from the user.

As one of the intentions behind the GCI is to make it possible to switch GC algorithms without changing the application code, explicit locking instructions are problematic, as the level of synchronization required depends on GC algorithm and implementation. E.g. a copying

or compacting collector needs locking at both read- and write barriers, while a non-moving mark-sweep only has a write barrier. Also, depending on how root operations and function calls are implemented, the required synchronization varies. This could be solved by having a number of different locking instructions for different operations (for instance, `gc_lock_READ()`, `gc_lock_WRITE()`, `gc_lock_ROOT()`, `gc_lock_CALL()`, `gc_lock_RETURN()`, etc.) which would increase the complexity of the interface, and the risk of programming errors, significantly.

It would be possible to add a gc-lock optimization pass to the LJRT compiler by e.g. performing analysis similar to the PMH placement in [ACM+03]. However, explicitly placing lock instructions in the application code has the drawback that their placement must depend on the intended target platform and desired timing properties. That is, the programmer, or tool, must know that, for instance, a piece of code must have critical sections that are less than 10 μs *on a particular computer*. Thus, the analysis may need to be drastically different if we compile for a small 8-bit 8 MHz micro-controller or a 2 GHz machine.

From our point of view, flexibility and portability are very important, and therefore we believe that the application code should be as generic as possible, and that low-level decisions regarding the real-time behaviour and scheduling should be left to the run-time system. This is motivated not only by portability but also by the fact that much more information is available at run-time than statically at compile-time.

7.4.2 A read barrier requires locking

Copying and compacting garbage collection algorithms have, among other things, the advantages that fragmentation is avoided and that allocation is a constant time operation. This, however, comes at the cost that a read barrier is required and this is potentially expensive. In our implementations, the extra cost of the read barrier itself compared to a non-moving GC is just an extra pointer dereference for each reference access. In the context of concurrent GC, the main cost of the read barrier is that it requires synchronization to avoid that the collector moves an object while it is being accessed by the mutator.

The total synchronization overhead incurred by the read barrier has two parts; the cost of each lock/unlock operation, and the number of accesses. Typically, variables are read much more often than they are written, (or at least as often, for most programs) and thus, any overhead associated with the read barrier has a higher (or at least as high) impact on overall performance than the write barrier overhead. This means

that, for any application, the locking overhead will be at least twice as high when using a moving collector compared to a non-moving.

7.4.3 Locking at method calls

Passing references as parameters to a function may need to be protected. For instance, if a reference can exist only as a parameter (e.g., as in `foo(new Bar())`), the GC must not run until the parameter has been rooted (registered as a root) in the called context.

For functions returning references, it is possible (or likely) that the object to be returned has been allocated in the called function. Therefore, when the function returns, the only reference to that object is the return value, which may have been referenced only by a local variable. This means that the return value must be protected, both to make sure that the object is retained and scanned, and to prevent the GC from moving the object until a proper reference has been rooted and the write barrier executed in the calling context.

7.4.4 Effects on optimization

Another important aspect of synchronization, which is not addressed here, is the interaction with an optimizing back end. For instance, it can make a big difference if the lock/unlock instructions can be inlined by the compiler or if they have to be function calls. Specifically, if the synchronization instructions break up basic blocks, this would severely limit an optimizing compiler's options.

7.5 Reducing the overhead

This section outlines some observations that can be used to drastically reduce the overhead associated with heap locking. It is shown how the cost of function calls and root operations can be significantly reduced and how the overhead can be almost completely eliminated for highest priority threads.

7.5.1 Reducing the need for synchronization

The level of required synchronization is affected both by the choice of GC algorithm (e.g., if a read barrier is required or not) and by different implementation decisions in the compiler and run-time system. This

section gives examples of how those issues can be addressed in the compiler and in the run-time system, respectively.

Root alias analysis In a typical object oriented program, a large part of local variables will be of reference types, and thus there will be many root references. A special case is the temporaries used in LJRT programs. As stated, the GCI requires complex constructs, such as `foo = a.b.c`, to be split up into simple attribute accesses as shown in Figure 7.5. This means that a lot of roots has to be pushed on and popped from the root stack, causing a significant execution time overhead, primarily from the required synchronization.

It can, however, be observed that in order to ensure correct GC behavior, it is enough that each live object is reachable from one root[5]. This means that the amount of necessary root operations, and thereby the overhead, can be reduced; if it can be statically determined that a variable will only reference objects that are also referenced by another variable with longer lifetime, the "inner" variable does not have to be registered as a root. We call this *root alias* analysis, and the compile-time analysis is trivial, as we do whole-program compilation. With this optimization, the push and pop operations in Figure 7.5 would be removed, which means that there will be no additional overhead of having the temporary variables explicitly in the code. In a typical Java program, the amount of "root duplication" is, in our experience, very high, as the associativity between objects tend to be high — between $50\,\%$ and $70\,\%$ of roots (including temporaries) were found to be statically redundant in our experiments. A large portion of the local variables that really need to be rooted are temporary references required to keep a newly allocated object live before its constructor has completed. This is needed to keep latency low; as the constructor can be of arbitrary length it cannot be treated as atomic.

As an example of how the root alias analysis works, we take the code fragment in Figure 7.9. There, `f` and `b` will (or may) reference objects that are allocated in the context of `main`, so these variables must be registered as roots, as they are the only references to the new objects. On the other hand, in `proc`, we know that the parameters have been registered as roots in the calling scope. Local analysis in `proc`, can statically determine that `t1` and `t2` only reference objects that are reachable from (the

[5]This does not hold for moving collectors that use forwarding pointers in the objects, as the roots are used for updating pointers as well as for finding live objects; for this optimization to work, the read barrier must be implemented using an indirect table outside the object.

```
void main() {
     Foo f; Bar b;
     ...
     f = new Foo();
     b = new Bar();
     ...
     proc(f,b);
}
void proc(Foo foo, Bar bar) {
     Test t1, t2; Bar b1;
     ...
     t1 = foo.test1;
     t2 = foo.test2;
     b1 = bar.x();
     ...
}
class Foo {
   Test test1, test2;
   ...
}
class Bar {
   Bar b;
   ...
   public Bar x() { return b; }
}
```

Figure 7.9: *Root alias example*

attributes of) the parameters, and therefore it is not necessary to register these variables as roots. In contrast, we cannot tell if `b1` is an alias for something already rooted, or not. By analyzing the method `Bar.x()` it is seen that `x` only returns an object reachable from an attribute. Therefore, `b1` does not need to be registered as a root.

If we are doing whole-program compilation, all calls to functions returning references can be analyzed and will finally boil down to either an attribute access (which doesn't require rooting) or an allocation (which does). In a separate compilation context, it is not generally possible to perform the whole-program root alias analysis, but the local analysis may still be used to get rid of unnecessary roots caused by temporary variables.

The implementation of the root alias analysis is quite simple, and the majority of the code is shown in figure 7.10. This is code written for JastAdd II, an aspect-oriented compiler compiler tool [Ekm04, NIEH04], but it is basically Java code for evaluating the attributes of syntax tree nodes. For example, the first method describes the evaluation of the

`isNewRoot` attribute in `VariableDeclaration` nodes, which will evaluate to **`true`** if there is any statement that may cause the variable to contain a unique root.

In the case of class overloading, the analysis of whether a method call may return a new root must analyze all overloaded implementations of the method which may be executed, which may yield a conservative result. For the sake of readability, that code has been left out from the figure.

```
boolean VariableDeclaration.isNewRoot() {
  boolean result = false;  Stmt stmt = null;
  ASTNode scope = getSurroundingScope();
  foreach stmt in scope {
    result |= stmt.isNewRoot(this);  }
  return result;
}
boolean ExprStmt.isNewRoot(VariableDeclaration varDecl) {
  if (getExpr() instanceof AssignSimpleExpr) {
    AssignSimpleExpr expr = (AssignSimpleExpr) getExpr();
    return expr.getDest().isUse(varDecl) &&
           expr.getSource().isNewRoot();  }
  return false;
}
boolean MethodAccess.isNewRoot(){return decl().isNewRoot();}
boolean VarAccess.isNewRoot(){return decl().isNewRoot();}
boolean MethodDecl.isNewRoot(){ return returnsNewRoot();}
boolean InstanceExpr.isNewRoot(){return true; }

boolean Block.returnsNewRoot() {
  boolean result = false;
  for (int i=0; i<getNumStmt(); i++) {
    result |= getStmt(i).returnsNewRoot(); }
  return result;
}
boolean ReturnStmt.returnsNewRoot() {
  boolean result = false;
  if (hasResult()) { result = getResult().isNewRoot(); }
  return result;
}
boolean MethodDecl.returnsNewRoot() {
  // Native methods do not have bodies, so let's be conservative
  boolean result = true;
  if (hasBlock()) { result = getBlock().returnsNewRoot(); }
  return result;
}
```

Figure 7.10: *Root alias analysis in the front-end*

Function calls For function calls, the level of locking required depends on how reference arguments are passed — as references or as actual pointers (i.e., if the read barrier is executed in the caller or in the callee). In our implementation, reference structures are stack allocated and thus will not be moved by the GC. Therefore, if references are called by reference (i.e., a pointer to the reference structure is passed) no new roots are pushed in the callee and no heap locking is required. As the caller will always out-live the callee, if parameters to functions are known to be rooted in the calling context they don't have to be rooted again in the called context. Similarly, we know that the return value of a function will be used in the calling function (or not at all). Therefore, the variable that will receive the return value must already be rooted so if we pass a reference to this variable to the called function, it can be assigned before the return which removes the need to protect the return value. If function arguments and return values are handled in this way, no locking is required for function calls.

Root stacks in multi-threaded programs Another example of overhead caused by an uncooperative environment is the root stacks. In multi-threaded programs, each thread has its own root stack, and therefore, all root operations (i.e. push and pop) requires a pointer to the root stack of the current thread. In a system where the thread scheduler is Java-aware, the root stack pointer is part of the execution context of each thread and is saved and restored automatically.

In systems which cannot rely on scheduler cooperation, this has to be handled in the application code. As the root operations are part of the application code, and the current thread is not known at compile time, this must be looked up at run time. Looking up the root stack at each root operation is quite inefficient so this should be done once for each function call and cached. Similarly, if no root operations are done in a function (like in e.g. a typical math function of the standard library), such lookup is unnecessary. Therefore, lookup of the thread root stack is done lazily at the first root operation of each function and the result is cached. This can be implemented quite efficiently.

Highest priority threads If a thread is known to have the highest priority it will never be preempted by another thread during its execution. Therefore, it is enough to lock the heap (or rather, ensure that no other thread has locked the heap) each time such a thread starts executing. For a periodic thread, this could be implemented by placing a `gc_lock();` at the start and a `gc_unlock();` at the end of each sample. This almost

completely removes the locking overhead for the (set of) highest priority thread(s) without affecting the real-time behaviour of the application[6].

Furthermore, it enables much more aggressive optimizations to be applied to the code of HP threads, as it is known that no GC can occur during execution and the heap only needs to be in a consistent state when the HP thread stops executing. This means that also a part of the read and write barrier calls can be removed, reducing the inlined overhead and, as another consequence, allows more optimizations in the back end, at the machine code level.

This assumes independent threads, which is a reasonable assumption for the high priority threads in a control system. If a thread contains blocking calls (e.g., semaphore or monitor operations) the heap must be unlocked before each such call, or there will be a risk of deadlock.

7.5.2 Reducing the cost of synchronization

With fine-grained memory operations and heap-intensive applications, such as Java programs, the heap is almost always locked, so whenever preemption occurs, the probability that the heap is locked is high. Assume that a thread (`T1`) is executing and is in the middle of a memory operation. Then, a context switch occurs; the thread that is scheduled to run (`T2`) will probably try to lock the heap very soon after the context switch and be blocked. Then `T1`, which is holding the heap lock, is scheduled to run again until it releases the heap lock, allowing `T2` to continue its execution. This means that there will be three context switches instead of one, increasing the execution time overhead due to such *context switch chatter*.

Low latency due to locking is a requirement, so just increasing the size of the critical sections is not a viable solution. Therefore, we need a solution that allows very fine-grained preemption without the overhead of frequent unlocking and re-locking. We also need to make sure that context switches are not performed when the heap is locked.

This section will sketch three possible solutions based on turning off interrupts, preemption points, and a proposed technique, lazy locking, respectively.

Turning off interrupts The straight forward solution is to simply implement `gc_lock()` by turning off (clock) interrupts and `gc_unlock()`

[6] Assuming that no preemption takes place between threads of the same priority, as is the common case in real-time systems. This is no restriction, as if the system is schedulable the ordering between threads of the same priority doesn't matter

by turning them on again. On most architectures, interrupt requests that arrive when interrupts are masked are latched, so that when the interrupts are turned back on, any missed interrupt will be generated and the corresponding interrupt routine is executed. On such an architecture, this will give the desired semantics that if a time-slice ends, and preemption should take place, when the heap is locked, the context switch is delayed until the heap lock is released. Turning off interrupts may, however, not be allowed by the OS, or have negative effects on other parts of the system, e.g., interrupt-based drivers for peripherals, etc.

Preemption points By using a scheduler which only allow preemption at certain, pre-determined points, we can avoid frequent locking/unlocking. In fact, if the memory accesses are taken into account when placing preemption points so that preemption is only allowed when the heap is in a consistent state, no additional housekeeping or synchronization is needed in order to ensure correct GC operation.

Preemption points are problematic for two reasons. The first is that most standard real-time operating systems don't support them. The second one is that calling external native code (that doesn't have preemption points) may cause priority inversion. An illustrating example is a background thread calling an external routine with a long execution time. As external code doesn't have preemption points, high priority threads may be delayed indefinitely. One solution is switching to "native" preemption when calling external code and then switching back to preemption-points when executing known code. Drawbacks include a more complex scheduler implementation, and increased latency for external code due to the extra housekeeping required. The latter may not be acceptable if the external code is run in timing-critical parts of the application, e.g. if the external code is a controller generated from a simulink diagram or low-level legacy code.

Lazy locking If turning off interrupts or using preemption points is not possible or desirable, an alternative strategy for reducing the locking overhead is based on the observation that, while the frequent locking and unlocking is required in order to achieve low latency, in the common case, the heap is unlocked, and then shortly re-locked by the same thread. Thus, most of the locking operations are really unnecessary and most unlock–lock pairs could be removed without changing the behavior of the program (other than reduced overhead). The problem is just determining which lock and unlock operations that need to be performed. This could be done statically, but the analysis would be difficult

and highly dependent on the low-level scheduling, control flow based on input data, etc. Therefore, a dynamic, on-line approach is preferable.

For example, take a code sequence like in Figure 7.11. If we are executing in the marked region, and no clock interrupt has arrived (i.e., the thread will not yet be preempted), it is unnecessary to perform the unlocking and re-locking operations. Thus, if we could dynamically decide whether to perform the unlock/lock operations (in a way that is much cheaper than actually performing the locking), the overhead could be reduced. Then, when a clock interrupt occurs, the heap should really be unlocked at the next unlock instruction and the context switch performed.

```
          gc_lock();
          ...
-->       gc_unlock();
-->       gc_lock();
-->       ...
-->       gc_unlock();
-->       gc_lock();
          ...
          gc_unlock();
```

Figure 7.11: *Locking example: Small atomic operations cause frequent locking.*

One way of implementing this is by having two versions of the operations: the actual lock/unlock operations (which are executed when the locking is required) and "NOP" versions that are used when unlocking and re-locking isn't necessary. Then, the run-time system ensures that the correct version is run at each time to both guarantee the correct semantics and achieve the best performance. In principle, an implementation of this scheme looks like in Figure 7.12. This method gives similar behavior as preemption points with regard to heap accesses, but without requiring additional housekeeping in order to allow external native code to be run with real-time guarantees.

In the sketched implementation, the `reschedule` function in the scheduler is modified to include the lazy locking related operations. If modifying the scheduler is not possible, or practically feasible, much of the benefit of lazy locking can still be obtained if the OS has a call-back hook for a method to be called at context switches. In fact, this is the method used in our Linux/RTAI prototype, and it gives the same reduction of the number of locking operations, but does not address context switch chatter. That may, however, be a reasonable trade-off for not having to modify the scheduler.

```
void (*gc_lock)(void);
void (*gc_unlock)(void);

void gc_lock_real(void)
{   lock(heap_mutex);
    gc_lock   = f_nop;
    gc_unlock = f_nop;
}
void gc_unlock_real(void)
{   unlock(heap_mutex);
    yield();
}
void f_nop(void) { return; }
void reschedule(void)
{   if(is_locked(heap_mutex)) {
       gc_lock   = gc_lock_real;
       gc_unlock = gc_unlock_real;
     } else {
        /* perform actual context switch */
    }
}
```

Figure 7.12: *Lazy locking implementation sketch*

There are, of course, many other small details that must be taken care of when implementing such a scheme; e.g., the system must ensure that the heap is always unlocked before a blocking call is made or before a thread dies; otherwise there is a risk of deadlock.

7.5.3 Compiler optimization effects

Another problem with locking is that the lock/unlock operations are function calls or inline assembler, and that tend to break basic blocks and interfere with compiler optimizations. This is, partly, intentional as many optimizations are not safe in the general case. For instance, we must make sure that pointers (gotten through the read barrier) to objects are always read from memory. Otherwise, objects may have been moved since the last access, and such race conditions will lead to memory corruption.

However, preventing such optimizations is really only needed when a context switch actually has taken place; as long as the same thread is executing, any optimization is legal, as long as the heap and all references in memory are consistent at the next context switch. Thus, performance could be improved significantly if it was possible to implement

lazy locking in a way that the fast case did not break basic blocks. We believe that this could be done with self-modifying code, injecting the lock/unlock operations into the code where they are needed and modifying the lock/unlock instructions so that they ensure heap consistency. This, of course, requires detailed information about the inner workings of the optimizing back-end and target architecture and cannot be done in a simple or portable way.

7.6 Summary

An implementation of a framework for accurate, concurrent, real-time garbage collection aimed at embedded systems was presented. It allows very low latency and works for automatically generated C code, a standard C compiler and a standard real-time operating system, and we have evaluated its performance in a robotics application. The results show that it is possible to use accurate garbage collection in an uncooperative environment for real-time applications which require latency times as low as a few microseconds.

However, due to the restrictions imposed by the uncooperative environment — especially scheduler — explicit synchronization between mutator and collector is required, and this adds to the execution time overhead of memory operations. That means that we get a trade-off between performance (throughput) and predictability (latency) since if we require low latency the critical sections must be small, and that, in turn, requires more frequent synchronization. The synchronization overhead must also be taken into account when choosing GC algorithm; e.g., a copying or compacting GC requires a read barrier (which requires synchronization) and this may have a big impact on throughput as reads are typically much more common than writes.

Further, optimizations, in both the Java compiler and in the GC implementation, aimed at reducing both the cost of and need for synchronization was presented. Experiments show that the overhead can be significantly reduced without affecting the worst-case latency.

It is concluded that concurrent GC is feasible for use in hard real-time systems, even in an uncooperative environment. The run-time overhead can be kept at a reasonable level, and that cost may in many cases be acceptable in order to get the safety and predictability of accurate GC also in hard real-time threads.

CHAPTER 8

EXPERIMENTS

This chapter presents experimental support for the proposed techniques. After a brief presentation of the applications and execution environments, experimental support for the presented techniques is presented. Section 8.2 presents experiments with time-triggered GC and shows how using different scheduling methods affect the scheduling of GC work. Auto-tuning of the GC cycle time is studied in Section 8.3, and Section 8.4 presents experiments illustrating the different approaches to GC execution time estimation. Section 8.5 shows how priorities for memory allocations can improve robustness and performance. Section 8.6 contains experiments with memory-aware feedback scheduling. The performance of the LJRT run-time system, illustrating an accurate GC in an uncooperative environment, is examined in Section 8.7.

8.1 Experiment platforms

The experimental verification has been carried out in two control applications, the ball-and-beam, a simple control process, and motion control of industrial robots. The execution platform has been the IVM virtual machine [Ive03] and natively compiled Java using the LJRT platform.

The applications were chosen since we need benchmarks that are representative for the kind of systems that benefit from a low-latency GC, i.e. real-time control systems. Standard benchmark suites such as SPECjvm98 don't fit very well in this context because of their batch-oriented character. In batch programs, incremental and concurrent GC just adds overhead without yielding any benefit. Also, batch programs and embedded control systems typically have drastically different mem-

ory usage patterns; the former tend to build some data structure, do some computations on it, and then deallocate it, whereas the latter typically run "forever" in steady state.

Ball and beam process

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 8.1.

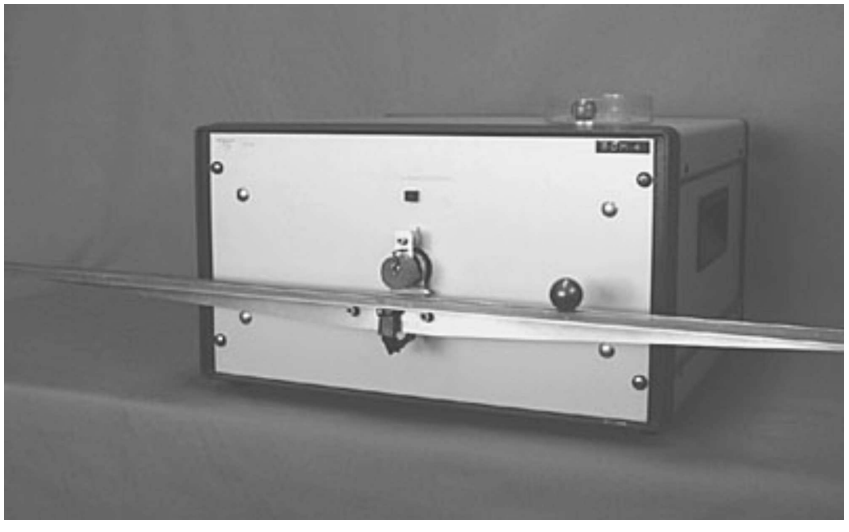


Figure 8.1: *The ball-and-beam process. The beam can be rotated to roll the ball to the desired position. Sensors measure the position of the ball and the angle of the beam.*

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 8.2. The reference generator and controller are run at a much higher rate than the UI thread.

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 opera-

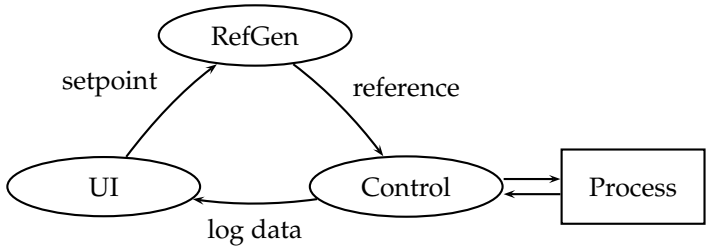


Figure 8.2: *The ball-and-beam control application consists of three threads; user interface, reference generator and controller. The data communicated between the threads is indicated by the arrows.*

tions 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 [NEN02] on a 350 MHz PowerPC and the memory allocation priority experiments were done using the IVM virtual machine [Ive03] on a STORK [AB91]/Linux platform.

Industrial robots

A recent master's thesis project [Lin04] made a Java implementation of the low-level servo controller for an ABB IRB-2000 industrial robot (given a desired motor velocity for each of the six joints, suitable torque values and the corresponding AC motor currents are calculated). Position samples and control signals are received and sent to the robot over a real-time network.

Also, a motion controller for an ABB IRB-6 was implemented. This is a standard PI controller. On the IRB6, local I/O on the control computer is used, making the control code simpler as sampling, calculation and output of the control signal are all performed in the same thread. With the exception of the drivers for the analog and digital I/O in the target system, the complete applications were written in Java. The IRB-6 controller was developed as a case study on the multi-stage deployment method presented in Section 2.4.3.

TrueTime-based memory management simulator

The proposed techniques for adaptive GC scheduling has been tested in a simulated environment. The simulations were carried out using TrueTime, a Matlab and Simulink-based system for studying embedded control systems by co-simulating the timing properties of a real-time kernel and the continous time dynamics of the process under control [HCÅ02]. On top of this, a simulator for a concurrent GC was implemented. The GC simulation is based on a generic heap model and a mark-sweep garbage collector. The heap model is driven by the mutator's allocation of objects and pointer assignmnent, and the GC is used to determine the number of live and dead objects (the mark routine) or to reclaim memory (sweep).

Based on the numbers and sizes of live and dead objects found by the mark routine, the amount of GC work required to complete a GC cycle is computed with a hand-written GC work function. That allows simulation of different GC algorithms by simply changing the work function. (currently, there is a mark-sweep, a mark-compact, and a copying collector). In each invocation of the GC task, the heap state is measured, and the GC work function is evaluated, and when the execution time of the GC task is equal or greater than the total work of that cycle, the cy-

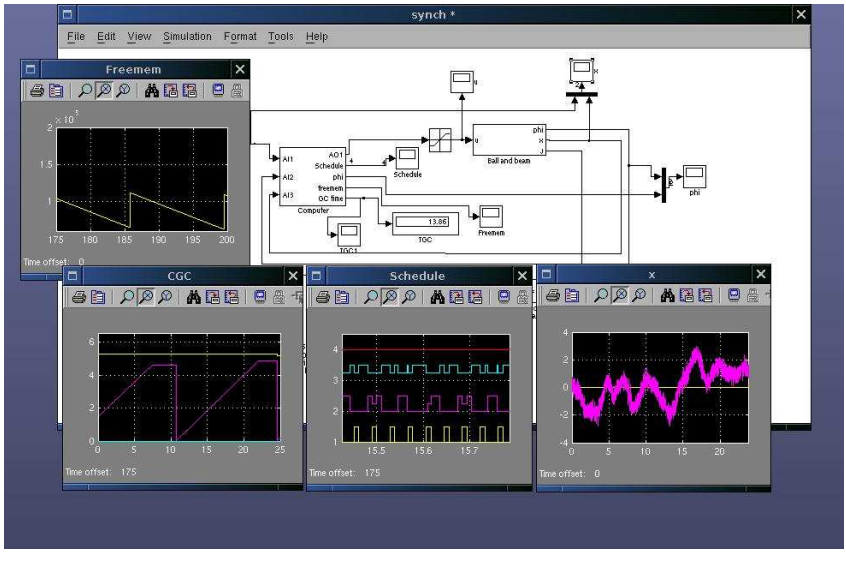


Figure 8.3: *Screenshot of the TrueTime-based memory simulator.*

cle finishes. Thus, the simulation is fairly accurate, as the actions of the mutator affects the CG workload of the current cycle. That also means that the scheduling will affect the amount of floating garbage — if the GC gets much CPU time early in the cycle, and finishes early, less objects will have had time to die.

Figure 8.3 shows a screenshot of the memory simulator. The physical process and the control computer are simulink blocks, and both the states of the process and different signals in the computer, such as the schedule, amount of free memory, GC cycle time and execution time, etc, are available as Simulink signals.

The application used in the simulations consists of two threads, a controller for the ball-and-beam, and a disturbance thread that generates garbage by allocating objects, filling and releasing buffers. It also causes transients by switching between operating modes with different allocation rates (varying size of allocated objects) and live memory amounts (varying buffer sizes).

8.2 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. 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).

Figure 8.4 shows an execution trace of a run with allocation triggered increments, in Figure 8.5 the same program is run with time-triggered GC with metric-scheduled increments and Figure 8.6 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 8.7 and Figure 8.8 show a close-up view of the thread graphs. Note 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 8.9 and Figure 8.10, and close-ups of the thread graph is shown in Figure 8.11 and Figure 8.12. 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. Notice that, by necessity, the cycle length of the time-triggered GC has been shortened in order to accommodate the higher allocation rate.

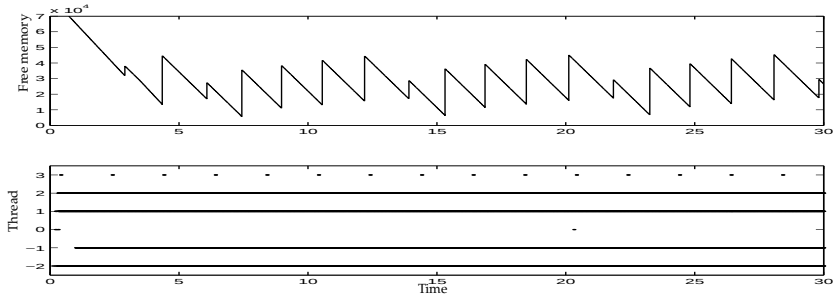


Figure 8.4: *Memory trace and schedule for the ball on beam application using allocation-triggered GC.*

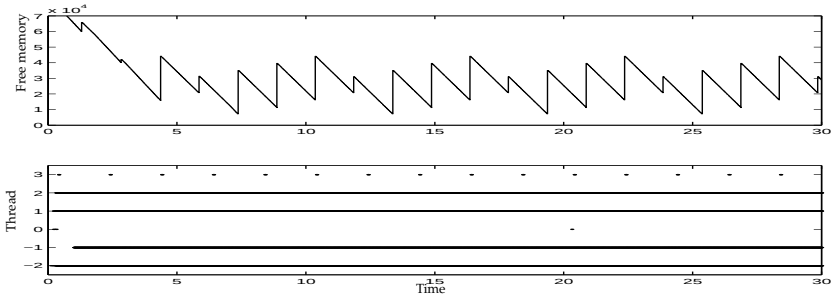


Figure 8.5: *Time-triggered with individually scheduled increments.*

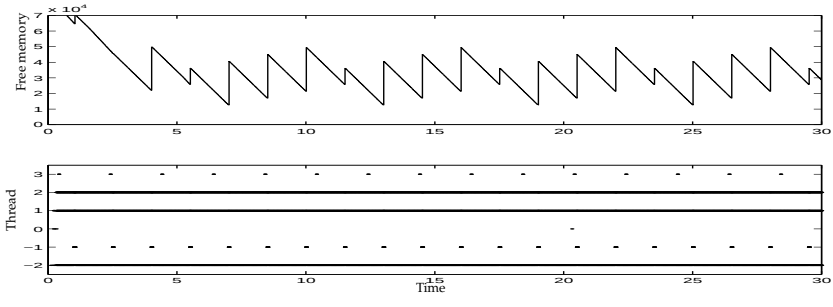


Figure 8.6: *Time-triggered, EDF scheduled.*

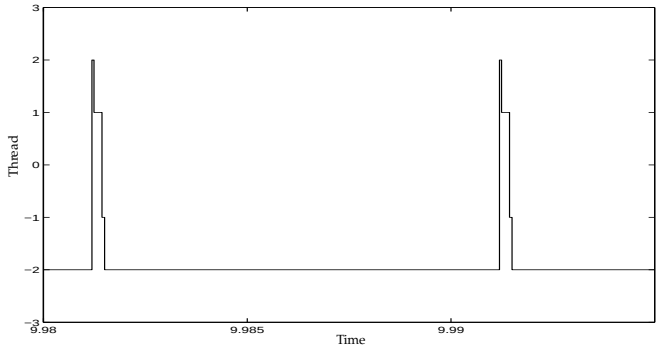


Figure 8.7: *Thread scheduling with the allocation-triggered GC. As the allocations performed during each thread period is small, the corresponding GC increment is also very short. The schedule of the time-triggered, metric-based scheduler is quite similar as both schedulers spread the GC work evenly 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.*

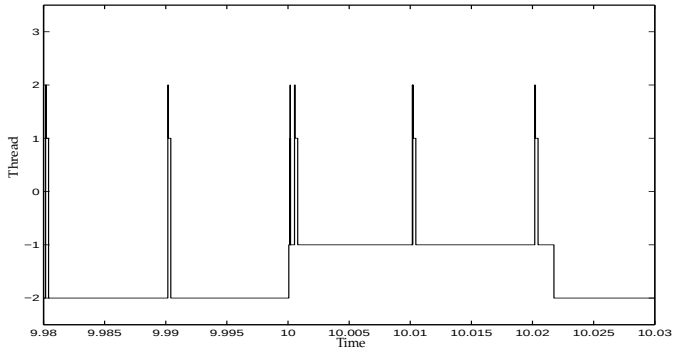


Figure 8.8: *Thread plot with the EDF scheduled GC. When a GC cycle is started, the garbage collector uses all idle time in order to perform the work required to finish the GC cycle as quickly as possible and then remains idle until the start of the next cycle. Each increment is, however, still very 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.*

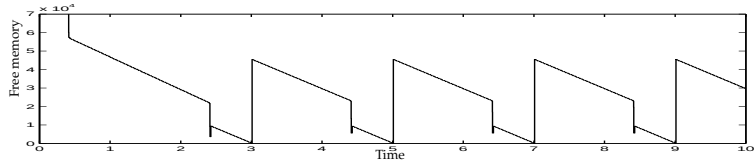


Figure 8.9: *Memory trace of an application with bursty allocations and allocation-triggered GC.*

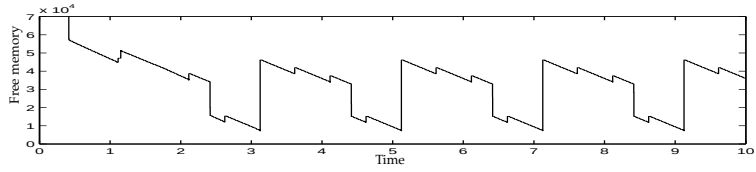


Figure 8.10: *Memory trace of an application with bursty allocations and time-triggered GC.*

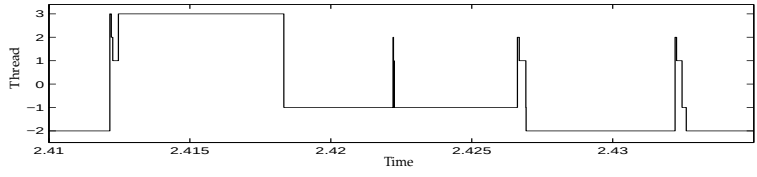


Figure 8.11: *Part of the thread graph corresponding to Figure 8.9. Note how a large allocation in thread 3 causes a long GC increment.*

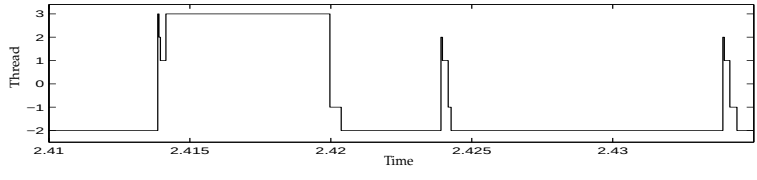


Figure 8.12: *Part of the thread graph corresponding to Figure 8.10. As GC work is not triggered by allocations, the GC work is spread evenly across the GC cycle, and long increments are avoided.*

8.3 GC cycle time auto-tuning

This section examines the adaptive GC cycle time estimates described in Section 4.2. Two sets of experiments are presented. The first one is an actual application executing on a real computer, and the second one was done in a simulator.

The first set of experiments show the ball-and-beam application running on the PowerPC/STORK platform, using EDF scheduling for threads. Figure 8.13 shows a memory trace of the system with the auto-tuner enabled. The fast threads run at 100 *Hz*. Figure 8.14 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 (4.3). Note the step in the T_{GC} graph near $t = 2.5$; no memory was freed during the first GC cycle, and therefore T_{GC} is halved.

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.

The second set of experiments were run in the simulated environment. Figure 8.15 shows how the T_{GC} tuner responds to changing allocation rates. Figure 8.16 shows the same experiment, using with the steady-state ΔG compensation as of Theorem 3. At the start, one of the threads run in an allocation-intensive mode with a random allocation pattern. At $t = 100$, it changes to a steady-state mode with a lower allocation rate, and at $t = 200$ it changes back to the random mode. Note that the ΔG compensation reduces the amount of reserved memory, when the mutator is in steady state, and how this reduces the variations in GC cycle times.

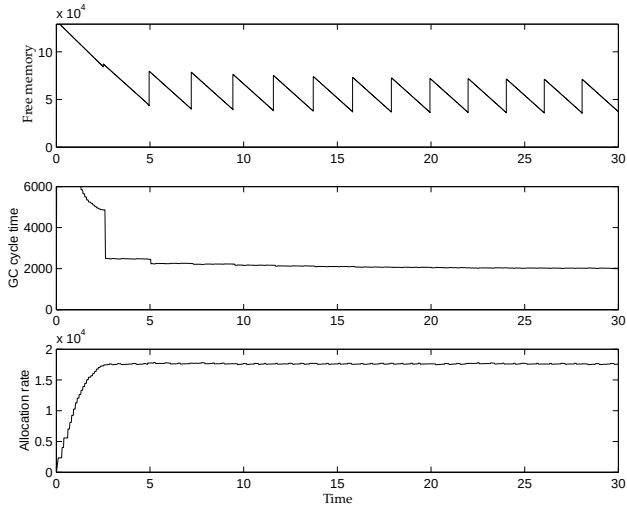


Figure 8.13: *Memory trace of the system with adaptive GC cycle length. The topmost plot shows the amount of available memory (in bytes), the middle plot shows the estimated GC cycle length (in milliseconds) and the bottom plot shows the LP filtered allocation rate measurement (in bytes/second).*

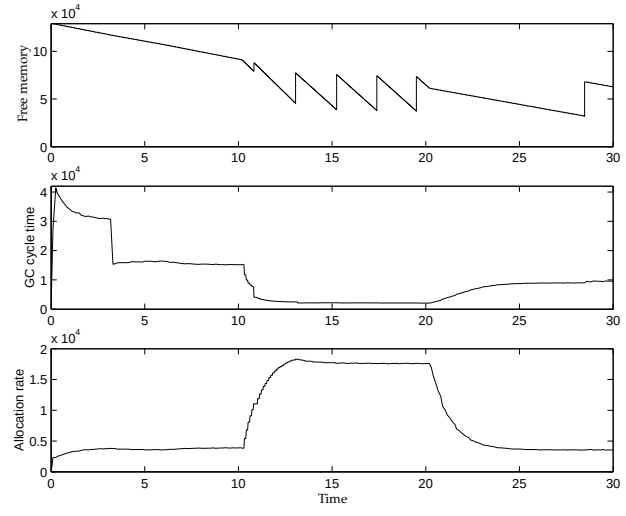


Figure 8.14: *How the GC scheduler reacts to changes in allocation rate; At* $t = 10$ *s, the frequency of the high priority threads is increased from* 20 *to* 100 *Hz and at* $t = 20$ *s the frequency is lowered to* 20 *Hz.*

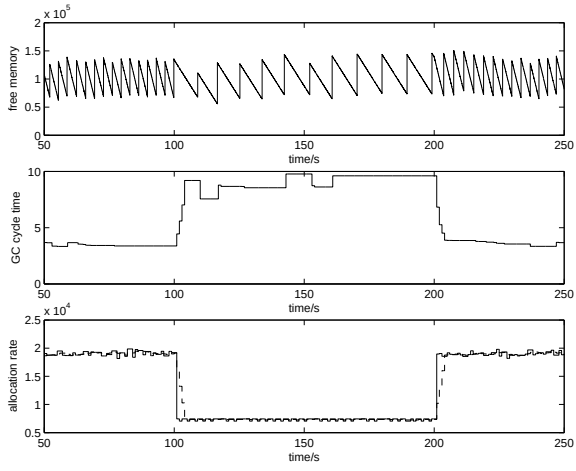


Figure 8.15: *Operation of the adaptive GC cycle length tuner. The topmost plot shows the amount of available memory (in bytes), the middle plot shows the GC cycle length (in seconds) and the bottom plot shows the allocation rate measurement (in bytes per second; the solid line is the actual samples and the dashed line shows the filtered measurement used for the T_{GC} calculation).*

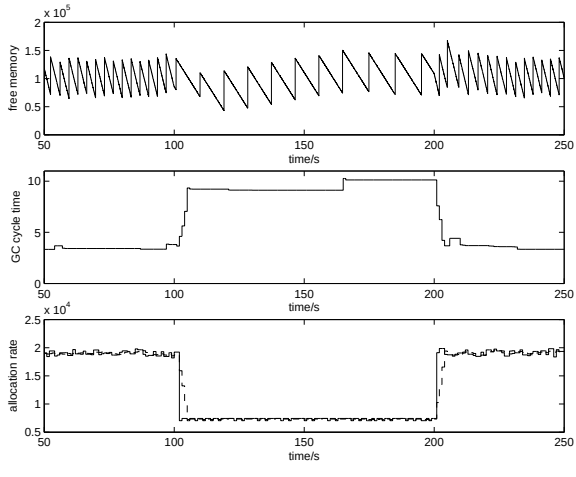


Figure 8.16: *Operation of the adaptive GC cycle length tuner with steady state ΔG compensation. Note how T_{GC} is held constant in spite of varying amounts of floating garbage in the steady-state phase.*

8.4 GC work prediction

This section examines the performance of the different approaches to C_{GC} estimation. The same simulated setup, as in the cycle time tuning experiments, was used. In these experiments, the change from the random allocation mode to the steady-state mode was at $t = 75s$. Note that, as a feedback scheduler was used, the period times of the mutator threads, and thus the allocation rates, differ. As the C_{GC} estimates are based on the history of the GC thread, at start-up, the system is run with default values for a number of GC cycles. Also, initial allocations performed by run-time system and application threads affect the amount of live memory and are therefore included in the simulation. No effort has been made to handle start-up of the simulated system in a graceful way, and in order to allow the effects of such transients to die out, the plots start at $t = 50\,s$.

Figure 8.17 shows the black-box approach, using the maximum value of C_{GC} of the last four GC cycles as the prediction. As no actual prediction is done, this method occasionally under-estimates C_{GC}.

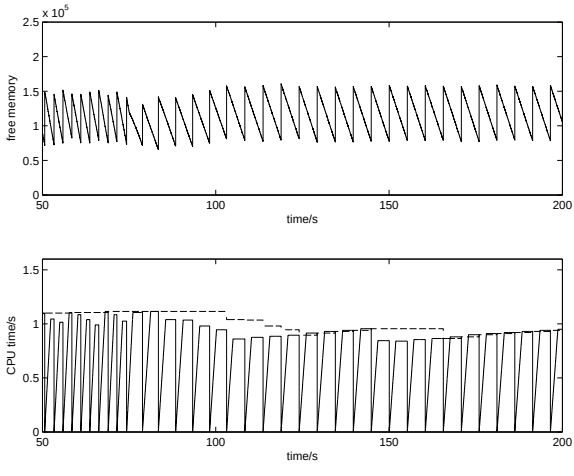


Figure 8.17: *A trace of the amount of free memory and the C_{GC} estimate using max of the last 4 cycles. In the C_{GC} plot, the solid line is the amount of CPU time spent on the current GC cycle, and the dashed line is the C_{GC} estimate.*

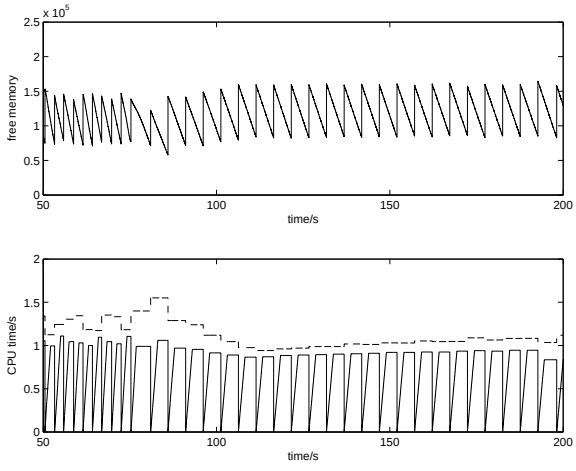


Figure 8.18: *C_{GC} estimate using $P(Live)$ and $P(Dead)$ according to Equation 4.33.*

Figure 8.18 shows the clear-box prediction. Note how the C_{GC} estimate is increased in the cycle after mode change, as the amount of memory on the heap is increased due to the ΔG compensation, but the estimates of $P(Live)$ and $P(Dead)$ are still at their old values (in this experiment, the forgetting factor was 0.9, meaning that old values decay with $10\,\%$ each sample.

Figure 8.19 shows the conservative prediction. The fraction of live memory was about 0.3, and the over-estimation was about a factor of two. In the steady-state mode, the actual U_{GC} was about 0.1, meaning that the over-estimation caused $10\,\%$ slack. In this experiment, the slack was not made available to the mutator, which can be seen in the visibly lower allocation rate (compared to the other two experiments), particularly during the allocation-intense phase, at the beginning.

As discussed, using a GC algorithm where live objects account for the greater part of the GC work, combined with a low fraction of live objects may cause large over-estimation of C_{GC}. This is illustrated in the example of Table 8.1, where the same application was run with different heap sizes. For this experiment, the application threads were scheduled with fixed period times (i.e., no feedback scheduling) in order to study the affect of the heap size on the C_{GC} prediction without having the prediction affecting the scheduling. It should be noticed that even if the conservatism increases as the fraction of live memory decreases,

the required GC utilization still decreases. Therefore, using a smaller heap to get better C_{GC} prediction is, in general, not a good idea. It can also be noted that the over-estimation in the experiment is less than the worst case conservatism. That can be explained by the fact that floating garbage makes the fraction of live objects – and, hence the GC work – larger than the ideal best case. This effect is exaggerated by using the same, fixed, period times for the application threads: as the GC utilization decreases, the slack in the schedule increases, allowing the GC to finish earlier and thereby causing more floating garbage.

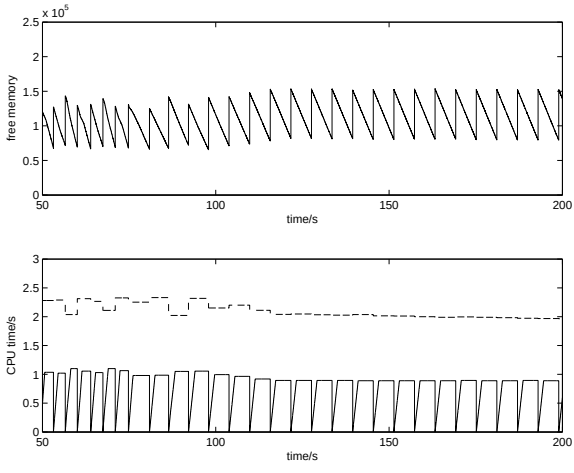


Figure 8.19: *Conservative C_{GC} estimate using Equation 4.35. In this experiment, $P(Live) \approx 0.32$ and $\frac{\alpha}{\beta} \approx 2.6$.*

Heapsize	250000	500000	2500000
$\hat{C}_{\mathrm{GC}}$	$1.5\,s$	$2.3\,s$	$10\,s$
C_{GC}	$1\,s$	$1.3\,s$	$3.9\,s$
T_{GC}	$15\,s$	$30\,s$	$160\,s$
$\hat{U}_{\mathrm{GC}}$	10%	7.7%	6.3%
U_{GC}	6.7%	4.3%	2.4%

Table 8.1: *Effects of heap size on the conservative C_{GC} estimation and GC overhead. While the degree of conservatism of the estimation increases as the heap size increases (i.e., as the fraction of live memory decreases), the total GC overhead (both estimated and real GC utilization) decreases.*

8.5 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. Experiments were run on the physical ball-and-beam process.

8.5.1 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 8.20 shows a run of the ball-and-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.

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.

[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.

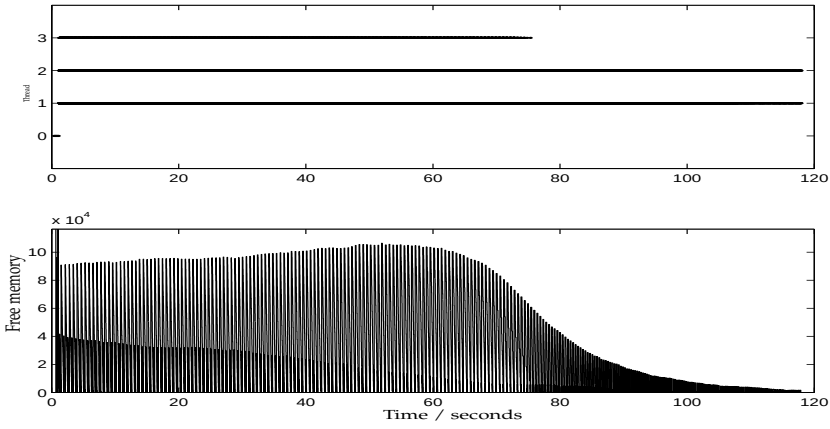


Figure 8.20: *A sample run of the ball-and-beam system without using memory priorities. The UI thread* (3) *doesn't get enough CPU time to consume all plot data that is produced. After $t = 75$ it is totally starved by the GC. Then, less and less memory is available and more and more CPU time is spent doing panic GC.*

Figure 8.21 shows the same system where the allocation of log data has been made non-critical, and the log data allocation is kept at a sustainable level. In this experiment, more than half of the log data allocation requests were allowed. Figure 8.22 shows a close-up of Figure 8.21 where you can see the non-critical behaviour more clearly.

8.5.2 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 8.23 show two runs of the ball-and-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.

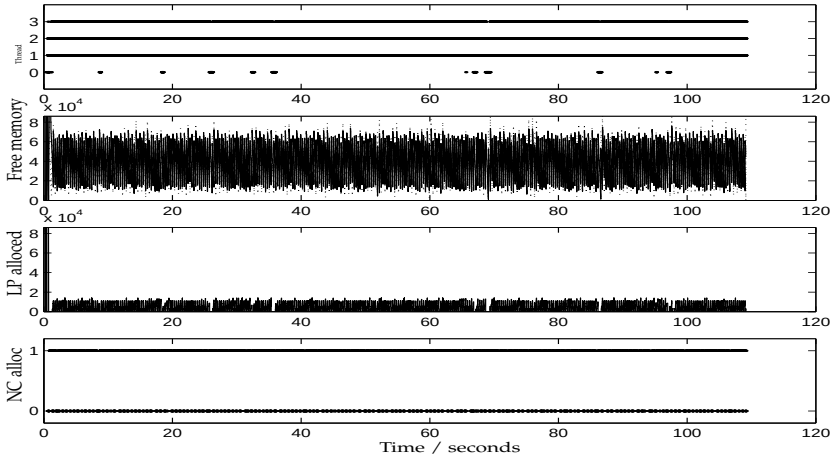


Figure 8.21: *A run of the ball-and-beam system with log-data allocations made non-critical. In the thread plot you see that the UI thread gets CPU time throughout the run. The third plot shows the amount of memory allocated by low priority processes during this cycle. The fourth plot shows if non-critical allocations succeed or not; high level means success and low level is deny.*

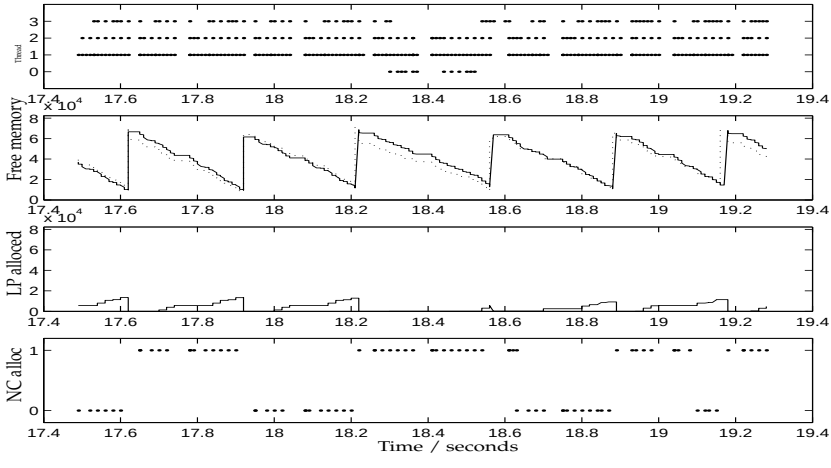


Figure 8.22: *Close-up to show the non-critical memory behaviour. The dotted line in the free memory plot is the non-critical limit. Note how the GC cycles are shortened when low priority allocations are made.*

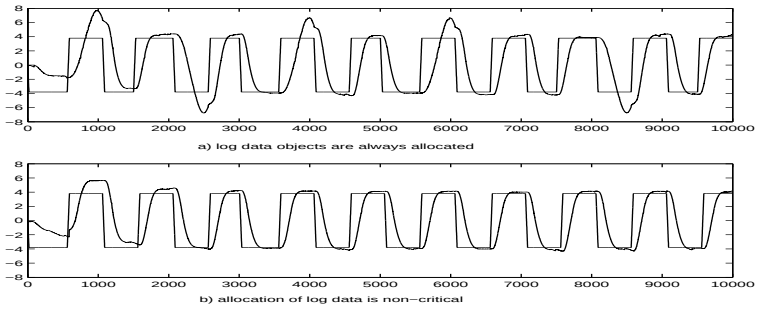


Figure 8.23: *Plots showing the reference value and the measured position for the ball-and-beam process. Plot* a *shows the system without non-critical memory allocations and plot* b *shows the system where the allocation of plot data is non-critical. The irregular behaviour in* a*, around samples 2500, 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.*

8.6 Feedback scheduling

This section presents simulations illustrating the behaviour of the different approaches to memory-aware feedback scheduling. The application is the TrueTime version of the ball-and-beam controller, and a disturbance task.

Three simulations are shown, with parameters and resulting control performance according to Table 8.2. Figures 8.24, 8.25, and 8.26 show the reference utilization for the mutator, and the sampling period of the controller task, under both the separate and the integrated feedback scheduler. In all plots, the solid line represent the integrated scheduler, and the dashed line is the version with separate GC tuner. The integrated scheduler used the simplified constraint, (6.17). As the approximation introduces errors, for a better comparison, the resulting period times of the integrated scheduler were scaled to get a mutator utilization of exactly U_{ref}.

These experiments show similar performance for both the separate and integrated versions. That supports the claim that the most important difference is the fairness issue, as in the integrated version, the amount of allocation affects the period assignment. This is apparent in Figure 8.26, where the difference in sampling period for the controller (h_1), differs more between the two schedulers, than in the other experi-

ments, due to the bigger difference in memory usage. It can also be seen that as the GC utilization decreases, the variation in U_{ref} also decreases.

Heap size	a_1	a_2	Cost, separate	Cost, integrated
100000	300	640	474	468
200000	300	640	473	426
200000	300	64	312	395

Table 8.2: *Parameters and control performance (cost, less is better) for the different experiments. a_1 is the allocation per sample of the controller, and a_2 of the disturbance task.*

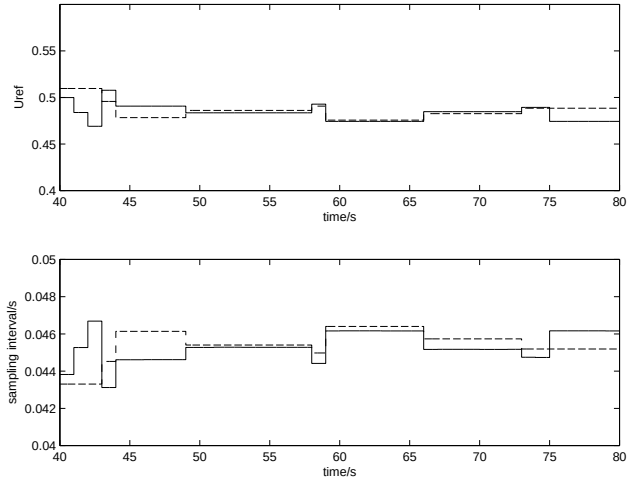


Figure 8.24: *U_{ref} and h_1, $a_1 = 300$, $a_2 = 640$, $H = 100000$*

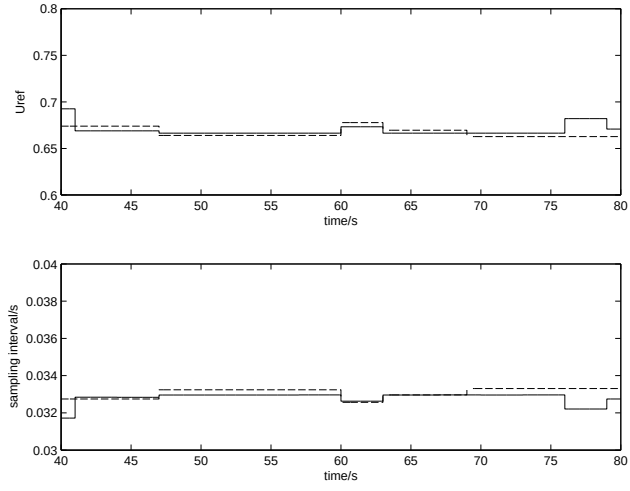


Figure 8.25: *U_{ref} and h_1, $a_1 = 300$, $a_2 = 640$, $H = 200000$*

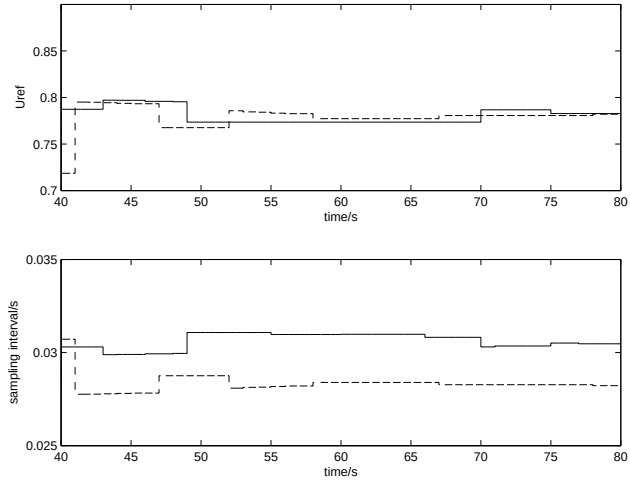


Figure 8.26: *U_{ref} and h_1, $a_1 = 300$, $a_2 = 64$, $H = 200000$*

8.7 Performance evaluation

In order to study the overhead due to GC in LJRT applications, the IRB-2000 slave controller was used. The program was compiled to C using the LJRT compiler, to native code with gcc and executed on a 350 MHz PowerPC G3 with 32 MB RAM running Linux/RTAI, and real-time performance and throughput (latency caused by memory operations and max possible sampling rate) was measured.

It should be noted that no attempt is made to compare the suitability of different GC algorithms for real-time systems; the implementations of the two collectors presented here are quite different, and should be considered as proof-of-concept prototypes, so a direct comparison is not meaningful. The aim of the experiments is to verify the feasibility of very fine-grained incremental garbage collection in an uncooperative environment.

8.7.1 Inlined overhead

This section studies how the choice of GC algorithm and the proposed optimizations affect the inlined overhead (e.g., read and write barriers, heap synchronization, GC housekeeping, etc) and, consequentially, throughput. Here, the inlined overhead is estimated by measuring the maximum possible sample rate.

The maximum sample rate measurements don't include the actual time to perform GC work, but only the inlined overhead, as the amount of time spent on GC work depends heavily on both the GC implementations and the allocation pattern of the application. Another reason for not taking the actual GC work into account in the throughput measurements is that our GC scheduling model is based on scheduling GC work so that it doesn't disturb the real-time tasks. Making sure that there is enough time for the GC to execute and meet its deadlines is a separate issue.

The sample rate measurements show that, in this application, the overhead of a moving collector is about twice that of a non-moving mark-sweep collector. The additional overhead comes from the additional locking for the read barrier. This is almost the best case as many of the objects in this application are very short lived (they only live for one sample) and only accessed once. Table 8.3 shows the number of locks per sample, the time spent in mutex operations and, based on this, the maximum possible sample rate if the actual computation took zero time.

Algorithm	locks/sample	time (μs)	max rate (Hz)
Non-moving	2980	715	1298
Moving	6088	1461	684

Table 8.3: *Frequency of heap locking and maximum possible sample rates based on lock overhead using mutex locking and no optimization.*

Figure 8.27 shows how the choice of locking primitive and the root alias optimization affects total throughput, i.e. the maximum possible sample rate. As a base line, using the batch-copy collector, without locking allowed 4317 Hz without the root alias optimization and 11038 Hz with. As the batch-copy collector doesn't have any read or write barriers, this also gives a rough indication of the performance that can be expected from applying the optimization of eliminating all GC synchronization for highest priority threads[2].

The big difference between the mark-sweep and the mark-compact collector is caused by the extra synchronization required for the read barrier in the mark-compact case. With synchronization turned off[3], there is no big difference between a moving and a non-moving collector. In this example, the overhead of the read barrier is compensated by the cheaper allocation[4]. For an application with a larger number of reads per write, the impact of the read barrier would be bigger.

8.7.2 Latency and jitter

In order to estimate how the GC synchronization affects thread latency, the `gc_lock()` and `gc_unlock()` instructions were instrumented to measure the time the heap was locked. The heap locking method used in this experiment was interrupt masking, which prevents context switches between `gc_lock()` and `gc_unlock()`. Thus, handling of any clock interrupt arriving when the heap is locked is delayed until `gc_unlock()` is executed. This delay is the added thread latency due to GC synchronization. Table 8.4 shows max, mean and median numbers for two GC algorithms. This gives an estimate of the worst case latency due to GC locking, which would occur if a high priority thread were released just after the heap was locked.

[2]This is a conservative estimate, as knowing that a HP thread will run uninterrupted also enables much more aggressive optimizations to be applied.

[3]Of course, running without synchronization is not safe and may cause race conditions and memory corruption, so this is done for reference only and is not practically usable.

[4]In the mark-compact collector, allocation is done by simply incrementing a pointer, whereas in the mark-sweep case, freelist search and block splitting is done.

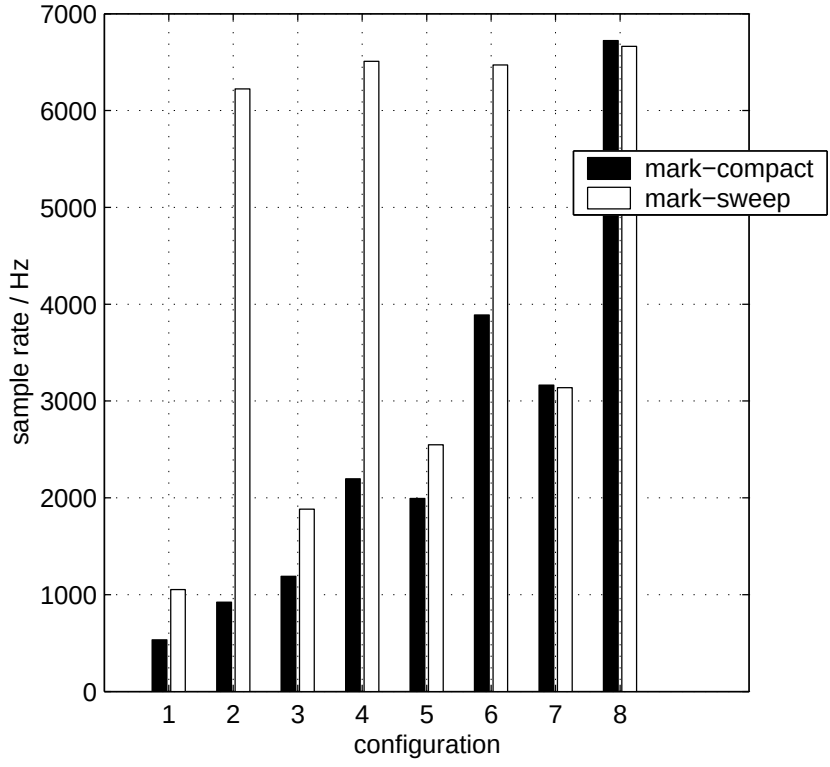


Figure 8.27: *The effect on throughput of different locking primitives and root optimization. The configurations are 1) Mutex locking, 2) Mutex locking with root alias optimization, 3) Lazy mutex locking, 4) Lazy mutex locking with root alias optimization, 5) Interrupt masking (cli/sti), 6) Interrupt masking with root alias optimization, 7) No locking and 8) No locking with root alias optimization*

The amount of added latency depends on how large the atomic operations is. Lower latency can be achieved by making the atomic operations smaller at the cost of more frequent locking and lower throughput. Keeping the locking time short is most challenging in the implementation of the GC, especially in a moving GC for which it is non-trivial to achieve shorter locking times than the time it takes to move one object. This is possible, however, either by detecting that moving an object failed [Hen98] or by limiting the size of individual heap objects [Sie02].

Algorithm	Max	Average	Median
Mark-sweep	12.3	2.7	2.6
Mark-compact	14.8	3.6	3.6

Table 8.4: *Heap locking times (μs) for the two GCs.*

A second experiment measured the actual latency of a thread with highest priority. In this experiment, the IRB-6 motion controller was used, as in that application the controller thread is explicitly periodic. Figure 8.28 shows the real-time performance of the controller thread, on the target system, and you can see that the jitter is quite low, typically below $\pm 3\,\mu s$.

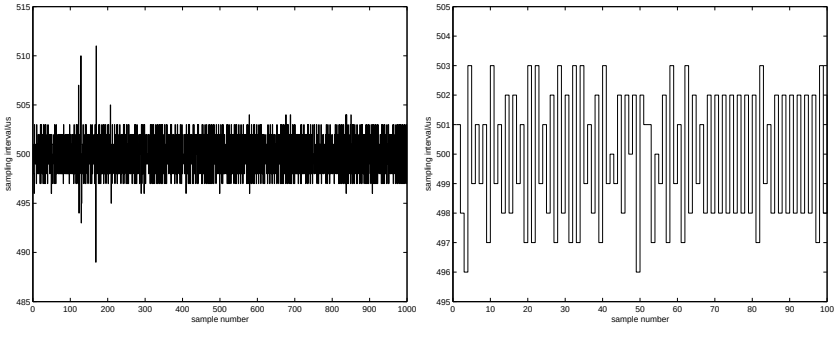


Figure 8.28: *Measured sampling intervals for 1000 consecutive sampling instants; the nominal sampling period was* $500\ \mu s$*, and the jitter was typically less than* $\pm 3\,\mu s$*, with a maximum of* $\pm 10\,\mu s$*. The right plot shows a close-up of the first 100 samples.*

8.7.3 Lazy locking

This experiment investigates the impact of lazy locking on the number of lock operations that are actually performed. Figure 8.29 shows the frequencies of locks in the vanilla version and real and lazy locks in the lazy version. This shows that only a small fraction of the locks actually need to be performed and thus that the locking overhead can be significantly reduced. For instance, in the receiver thread, which performs most of the computations, only 0.03% of the lock instructions in the code actually cause a mutex operation.

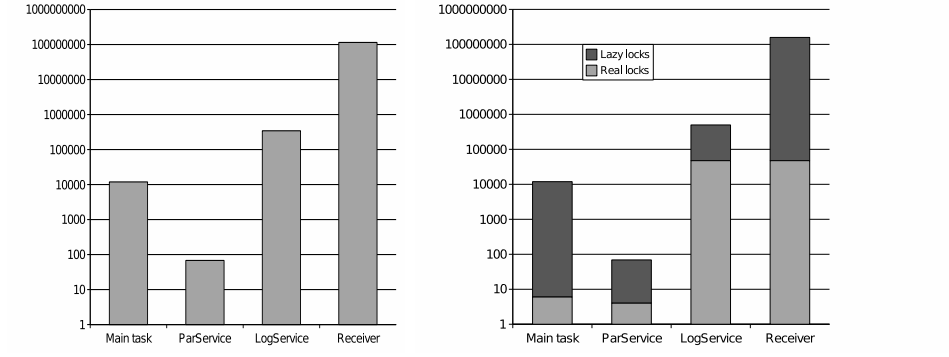


Figure 8.29: *Comparing the vanilla version (left) to the one with lazy locks (right), showing the frequencies of real and lazy locks for each of the application threads. Please note that the scale is logarithmic. In this experiment, the mark-compact collector was used. The application was run for a fixed amount of time; the vanilla version ran 17191 samples and the lazy version 23672 samples, so the numbers should not add up.*

CHAPTER 9

FUTURE WORK

This chapter outlines open problems and possible directions for future research in areas discussed in, or related to, this thesis.

9.1 Adaptive GC scheduling

The proposed approaches to C_{GC} prediction must be regarded as preliminary. Using a more detailed model of the heap state and predicting heap state by simulating a dynamic system was dismissed as not practically feasible. However, for application with small variations in memory usage and long periods between mode changes, and GC algorithms where different size distributions, etc., have large impact on the workload, it may be interesting to further explore that approach.

In the simplified clear-box workload prediction, the probabilities of live and dead cells are quite simplistic. Here it would be interesting to study how modelling the mutator as a stochastic process could improve the quality of the prediction.

The experimental evaluation of the presented techniques for autotuning GC scheduling has been limited to a small number of applications, and while they are representative for embedded control systems, a bigger set of benchmarks — including applications outside the field of automatic control — is desirable. The GC execution time prediction has to date only been tested in the simulated environment, and will require evaluation also in an actual run-time system.

An 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 when using an incremental-update collector. 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.

Another interesting situation is a system with a few hard real-time threads which requires a certain CPU percentage and a set of soft real-time threads. Then, after allocating the required CPU time to the hard real-time threads, the remaining CPU bandwidth should be divided between GC and soft real-time threads. Solving Equation 3.4 or 3.18 for a instead of T_{GC} would yield a safe allocation rate and hence, period time, for each low priority thread.

9.2 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.

In the presented work, only two levels of priority for memory allocation (critical and non-critical) are used. That has the advantages of being easy to handle, both at design time and in the run-time system, where the former is the more important. For the programmer, it makes the design decision quite clear: is a certain piece of code critical or not? Having more levels of priority would increase the power of expression of the model but, at the same time, make the meaning of a priority level less obvious. Nonetheless, an interesting direction of further study is whether there are applications where additional advantages may be gained from having an arbitrary number of memory priority levels.

9.2.1 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 or supervision, 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?

9.2.2 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[AOS]. 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 [KHH+01] to insert a test whether an allocation should be done before each call to a constructor.

9.3 GC scheduling interface

The experimental platforms were implemented using the garbage collection interface (GCI) [IBE+02] 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. When a black box approach to on-line GC scheduling 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.

9.4 Feedback scheduling and QoS

The results in Chapter 6 have only been tested in a simulated environment, and further evaluation on real implementations is required. In particular, the proposed idea of controlling the allocation rate of processes has to be developed further. In the presented case study, the change of allocation rate was implemented by switching between different controllers, and while suitable for some control systems, it is no general solution. This also motivates further research on how to make the behaviour of non-critical memory allocations configurable.

9.5 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 [Kop02, KB03] addresses the composability problem and its important features include 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.

For the same reasons, time-triggered GC scheduling might also be useful together with the linear control server model [CE03], which uses time-triggered I/O in order to avoid degraded control performance due to scheduling jitter.

CHAPTER 10

RELATED WORK

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

10.1 Time-based garbage collection scheduling

The fundamental idea of the presented work is that a deadline is assigned to the GC, and then the GC is scheduled as any other thread in the system using an arbitrary scheduling policy. I.e., as stated, the presented time-triggered approach to scheduling differs from other GC scheduling strategies (including the Metronome, the deferrable server approach, and semi-concurrent scheduling) in that it does not address the scheduling of the individual GC increments but leaves that to the normal process scheduler. As a consequence, the time-triggered approach contains no explicit rules for the relative priorities of GC and mutator threads.

However, in a typical application of time-triggered GC, the GC will seldom or never preempt mutator threads. If rate-monotonic scheduling is used, the period time of the GC will typically be much longer than that of mutator threads. If earliest deadline first is used, the deadline of the GC will be a relatively long time into the future most of the time. If there is some slack in the schedule, the GC will finish its work before being so close to its deadline that it actually preempts mutator tasks.

Henriksson

Using time as the GC work metric was discussed in [Hen98] 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, on such platforms, using time as the fundamental GC work metric is practically possible, and offers advantages over ad hoc metrics.

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 and their Metronome collector[BCR03b, BCR03a]. 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 the Metronome 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

[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 (3.20) behave as a time-based GC with small GC and mutator quanta.

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. Also, the Metronome time quanta are in the millisecond range, whereas the atomic operations of the collectors in the LJRT runtime system are a few microseconds. Here, the design goals behind their collector differ from the ones driving the work presented in this thesis; they focus on low overhead and consistent utilization while non-intrusiveness and low GC induced latency and jitter — possibly at the cost of higher inlined overhead — are the key issues behind this thesis.

Qian et al

Time-triggered GC was also proposed in [YSaSC02] 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.

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.

Xian and Xiong

Work on GC scheduling aimed at minimizing the memory requirement was presented in [XX05]. That approach is based on minimizing the response time of the GC, and thereby the memory required to satisfy allocation requests while the GC is running, and the technique used is to treat the GC as an aperiodic task and run it in a deferrable server [SLS95].

Embedded systems are typically quite predictable, including memory usage. Thus the GC really is a periodic task, and by making this explicit in the model, standard thechniques for scheduling periodic tasks can be used. Treating the GC as aperiodic and allowing it to interrupt mutator tasks at arbitrary times will lead to increased jitter.

10.2 Adaptive GC scheduling

An approach to adaptive GC scheduling aimed at minimizing the GC overhead is suggested by Henriksson [Hen96]. 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.

That approach is particularly useful if the difference between the worst and average case GC work is large, and the worst case is rare. As the conservative C_{GC} prediction presented here may, under some circumstances, be very conservative, a similar approach might be very useful in the applications discussed in this thesis. By using both a conservative estimate and a record of average U_{GC}, the GC could be scheduled according to the average case at the beginning of each cycle. Then, at a certain point in time (determined by the maximum allowed U_{GC}) if the GC hasn't finished its work, U_{GC} is raised to the maximum.

Engelstad and Vandendorpe [EV91] 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 [Sie02] 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.

10.3 Memory Management in Real-Time Java

There are two specifications for real-time Java; The Real-Time Specification for Java (RTSJ) [B$^+$01] and the Real-Time Core Extensions (RTCE) [JC00]. 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.

10.4 Soft references

The notion of non-critical allocation is somewhat related to the *soft references* found for instance in Java [J2S] 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 logical behaviour similar to 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.

10.5 GC in an uncooperative environment

The work presented here focuses on hard real-time systems for which we want to use standard C compilers and to run our application under a standard real-time operating system. This means that we have to find a way to deal with an uncooperative environment. The standard way to introduce GC in uncooperative environments is to use conservative garbage collectors such as the one devised by Boehm and Weiser [BW88]. Such GC algorithms are, however, not suitable for the types of real-time systems we are interested in, since we need the GC to be both accurate and predictable.

An algorithm which is accurate and which works well with standard C compilers is the one presented by Henderson [Hen02]. His aim was to refute the "common wisdom" that accurate GC is not possible without support from the compiler back-end by presenting such a GC for single-threaded applications and stop-the-world GC. The work presented in this thesis also illustrates that GC is possible without special compiler support. An important part of the contribution is to show that accurate

GC without support from the compiler or scheduler is possible also for concurrent GC and multi-threaded applications with strict real-time requirements.

10.6 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* [Sha89], for formalizing execution time analysis. A timing tool for a subset of C has also been developed [PS91].

In order to give continuous feedback to the developer, an interactive programming environment with worst case analysis functionality is desirable. The experimental tool Skånerost developed at our department provides interactive worst case execution time and memory consumption analysis based on timing schema and source code annotations for (currently a subset of) the Java language [PH99, Per99, PH00].

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 [EES01, Eng02].

Another approach to schedulability analysis and automatic verification of real-time systems based on timed automata has been developed in the UPPAAL project [LPY97, Pet99].

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 [Pus02]. 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.

For systems where threads execute with fixed period times, off-line assignment of GC scheduling parameters can be used. To facilitate this, a framework for performing static (compile-time) analysis of allocation rates was presented by Mann et al. [MDLC05].

CHAPTER 11

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 runtime 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 problems: to determine the scheduling parameters of the GC process and to keep a task set with varying resource utilization schedulable. Both problems were addressed. Methods for on-line estimation of both period and execution time of the GC were developed, and an approach to taking the CPU utilisation of the GC into account in a feedback scheduler were suggested.

Another approach to handling non-determinism and enhancing robustness, *applying priorities to memory allocation*, was presented. It was observed that often, systems contain parts that are not critical to the core functionality. Thus, 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.

11.1 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.

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, often, 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.

Memory-aware feedback scheduling

In order to use scheduled garbage collection in a feedback scheduling system, the required CPU utilization of the GC task must be known, and as the GC utilization depends on the memory behaviour of mutator tasks, it must be determined on-line. It was investigated how an auto-tuning time-triggered GC 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.

It was also suggested that non-critical memory allocations can be used, in a feedback scheduling system, to control the allocation rates of the application threads in order to optimize the trade-off between memory and CPU time usage.

Accurate real-time GC in an uncooperative environment

An implementation of a framework for accurate, concurrent, real-time garbage collection aimed at embedded systems was presented. It allows very low latency and works for a system including legacy and automatically generated C code, and an off-the-shelf compiler and operating system. Restrictions imposed by the uncooperative environment — especially the scheduler — makes explicit synchronization between mutator and collector necessary, adding to the execution time overhead of

memory operations. The sources of that overhead were analyzed, and possible remedies presented.

Experiments show that it is possible to use accurate garbage collection in an uncooperative environment for multi-threaded real-time applications which require latency times as low as a few microseconds, and that the run-time overhead can be kept at a reasonable level.

11.2 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 the real-time performance at run-time must be independent of *a priori* schedulability 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 will 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, help 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 task while still guaranteeing sufficient progress. This thesis shows that time-triggered GC scheduling has this property under both fixed priority and EDF scheduling.

The fundamental idea behind time-triggered GC scheduling is to turn garbage collection into a periodic activity that can be scheduled using standard scheduling techniques. It is my belief that adding a specialized scheduler for the individual GC increments merely adds overhead: if the system (including GC) is schedulable, there is no point in running the GC with a higher priority than the mutator — that just increases jitter. If the system is not schedulable it is better to explicitly limit the CPU usage of the mutator (using e.g. constant bandwidth servers, feedback scheduling, or some such approach) rather than running the GC at the highest priority with a specialized scheduler — the result is the same: the mutator threads are delayed and the GC is given enough CPU time to ensure sufficient progress.

On-line resource management through adaptive techniques and approaches to handling overload is an important part of the presented work. Based on the observation that not all hard real-time systems — or not all *parts* of a hard real-time system — are safety critical, techniques that transfer some resource-management tasks from the programmer to

the run-time system were proposed and discussed. While adaptive systems cannot give absolute guarantees, the presented mechanisms work together in enhancing robustness and providing isolation between different parts of a system. Their lack of hard guarantees can also be a strength as it forces the engineer to consider, and provides tools for managing, the uncertainty that is unavoidable in increasingly complex embedded systems.

Automatic memory management is essential to the use of safe object oriented languages and the presented contributions are a step towards making real-time garbage collection practically feasible.

BIBLIOGRAPHY

[AB91] 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.

[AB98] 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.

[ACM+03] Nevine AbouGhazaleh, Bruce Childers, Daniel Mossé, et al. Energy management for real-time embedded applications with compiler support. In LCTES'03 [LCT03].

[AEL88] 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.

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

[AP00] 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.

[AW89] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. Addison-Wesley, 1989.

[AW97] Karl Johan Åström and Björn Wittenmark. *Computer-controlled systems, Theory and design*. Prentice Hall, 1997.

[B+01] Greg Bollella et al. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[Bak78] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[BCR03a] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for Java. In LCTES'03 [LCT03].

[BCR03b] 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.

[BH04] Stephen M Blackburn and Anthony L Hosking. Barriers: Friend or foe? In *Proceedings of the 2004 International Symposium on Memory Management (ISMM'04)*, Vancouver, Canada, October 2004. ACM Press.

[BLA02] Giorgio Buttazzo, Giuseppe Lipari, and Luca Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3), March 2002.

[Bob68] D. G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 11(3), July 1968.

[BW88] Hans-J. Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software – Practice and Experience*, 18(9), September 1988.

[CE03] Anton Cervin and Johan Eker. The Control Server: A computational model for real-time control tasks. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003.

[CEBÅ02] Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1), July 2002.

[Cer03] Anton Cervin. *Integrated Control and Real-Time Scheduling*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, April 2003.

[CLE+04] Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Årzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Göteborg, Sweden, August 2004.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12), December 1960.

[DLM+78] 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.

[DN76] 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.

[EES01] 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.

[Ekm04] Torbjörn Ekman. *Rewritable Reference Attributed Grammars — design, implementation, and applications*. Licenciate thesis, Department of Computer Science, Lund University, 2004.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Department of Information Technology, Uppsala University, 2002.

[EV91] Steven L. Engelstad and James E. Vandendorpe. Automatic storage management for systems with real time constraints. In *OOPSLA '91 GC Workshop*, 1991.

[FY69] R. Fenichel and J. Yochelson. A lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11), November 1969.

[HC05] Dan Henriksson and Anton Cervin. Optimal on-line sampling period assignment for real-time control tasks based on plant state information. In *Proceedings of the Joint 44th IEEE Conference on Decision and Control and European Control Conference*, Seville, Spain, 2005.

[HCÅ02] Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002.

[Hen96] 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.

[Hen98] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[Hen02] Fergus Henderson. Accurate garbage collection in an uncooperative environment. In ISMM'02 [ISM02].

[HN99] Mathias Haage and Klas Nilsson. On the scalability of visualization in manufacturing. In *In Proceedings of ETFA '99*, 1999.

[IBE+02] 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.

[ISM02] *Proceedings of the 2002 International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, June 2002. ACM Press.

[Ive03] Anders Ive. *Towards an embedded real-time Java virtual machine*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2003.

[J2S] Java 2 platform, standard edition, API specification. Sun Microsystems. http://java.sun.com.

[JC00] J-Consortium. Real-time core extensions for the java platform. International J Consortium Specification, 2000.

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

[JP86] M Joseph and P Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5), 1986.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[KHH+01] 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.

[Knu73] Donald E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*. Addison-Wesley, 1973.

[Kop02] Hermann Kopetz. Time-triggered real-time computing. *IFAC World Congress, Barcelona, July 2002, IFAC Press*, July 2002.

[LC02] Bo Lincoln and Anton Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, NV, December 2002.

[LCT03] *Proceedings of the 2003 ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, San Diego, California, USA, June 2003. ACM Press.

[Lin04] Daniel Lindén. *Estimating the Overhead for Automatic Memory Management in Real-Time Systems*. Master's thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2004.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[LPY97] 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.

[Mac04] The Real-Time Java Platform, white paper. `http://research.sun.com/projects/mackinac/mackinac_whitepaper.pdf`, June 2004.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4), April 1960.

[MDLC05] Tobias Mann, Morgan Deters, Rob LeGrand, and Ron K. Cytron. Static determination of allocation rates to support real-time garbage collection. In *Proceedings of the 2005 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'05)*. ACM, 2005.

[Min63] M. L. Minsky. A lisp garbage collector algorithm using serial secondary storage. Memo 58 (rev.) Project Mac, M.I.T., Cambridge, Mass., December 1963.

[NEN02] 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.

[NIEH04] Anders Nilsson, Anders Ive, Torbjörn Ekman, and Görel Hedin. Implementing java compilers using ReRAGs. *Nordic Journal of Computing*, 11(3):213–234, 2004.

[Nil04] Anders Nilsson. *Compiling Java for Real-Time Systems*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2004.

[NR05] Anders Nilsson and Sven Gestegård Robertz. On real-time performance of ahead-of-time compiled Java. In *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC'05)*, Seattle, Washington, May 2005.

[Per99] 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.

[Pet99] Paul Pettersson. *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*. PhD thesis, Uppsala University, 1999.

[PH99] 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.

[PH00] 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.

[PS91] Chang Yun Park and Alan C. Shaw. Experiments with a program timing tool based on source-level timing schema. *Computer*, 24(5), May 1991.

[Pus02] 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.

[RH03] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In LCTES'03 [LCT03].

[Rit03] Tobias Ritzau. *Memory Efficient Hard Real-Time Garbage Collection*. PhD thesis, Department of Computer and Information Science,Linköping University, 2003.

[RNNH06] Sven Gestegård Robertz, Anders Nilsson, Klas Nilsson, and Mathias Haage. Multi-stage deplyoment of robot control software. In *Proceedings of the 8th International IFAC Symposium on Robot Control, SYROCO*, September 2006. to appear.

[Rob02] Sven Gestegård Robertz. Applying priorities to memory allocation. In ISMM'02 [ISM02].

[Sha89] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7), 1989.

[Sie02] Fridtjof Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 2002.

[SLS95] Jay K. Strosnider, John P. Lehoczky, and Lui Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Transactions on Computers*, 44(1), January 1995.

[SRL94] Lui Sha, Ragunathan Rajkumar, and John. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.

[Ste75] G. R. Steele, Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9), September 1975.

[Wad76] P. L. Wadler. Analysis of an algorithm for real time garbage collection. *Communications of the ACM*, 19(9), September 1976.

[Wel04] Andy Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, September 2004. ISBN 0-470-84437-X.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, France, September 1992. Springer-Verlag.

[WJNB95] 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.

[XX05] Yuqiang Xian and Guangze Xiong. Minimizing memory requirement of real-time systems with concurrent garbage collector. *ACM SIGPLAN Notices*, 40(3), 2005.

[YSaSC02] 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.