LICENTIATE THESIS

Programming Embedded Real-Time Systems: Implementation Techniques for Concurrent Reactive Objects

Simon Aittamaa

LULEÅ UNIVERSITY OF TECHNOLOGY

Programming Embedded Real-Time Systems: Implementation Techniques for Concurrent Reactive Objects

Simon Aittamaa

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

Supervisors:

Prof. Per Lindgren and Ass. Prof. Johan Nordlander

Printed by Universitetstryckeriet, Luleå 2010

ISSN: 1402-1757
ISBN 978-91-7439-194-7

Luleå 2010

www.ltu.se

Abstract

An embedded system is a computer system that is a part of a larger device with hardware and mechanical parts. Such a system often has limited resources (such as processing power, memory, and power) and it typically has to meet hard real-time requirements.

Today, as the area of application of embedded systems is constantly increasing, resulting in higher demands on system performance and a growing complexity of embedded software, there is a clear trend towards multi-core and multi-processor systems. Such systems are inherently concurrent, but programming concurrent systems using the traditional abstractions (i.e., explicit threads of execution) has been shown to be both difficult and error-prone. The natural solution is to raise the abstraction level and make concurrency implicit, in order to aid the programmer in the task of writing correct code. However, when we raise the abstraction level, there is always an inherent cost.

In this thesis we consider one possible concurrency model, the concurrent reactive object approach that offers implicit concurrency at the object level. This model has been implemented in the programming language Timber, which primarily targets development of real-time systems. It is also implemented in TinyTimber, a subset of the C language closely matching Timber's execution model. We quantify various costs of a TinyTimber implementation of the model (such as context switching and message passing overheads) on a number of hardware platforms and compare them to the costs of the more common thread-based approach. We then demonstrate how some of these costs can be mitigated using stack resource policy.

On a separate track, we present a feasibility test for garbage collection in a reactive real-time system with automatic memory management, which is a necessary component for verification of correctness of a real-time system implemented in Timber.

Acknowledgments

First of all, I would like to thank my supervisors, Per Lindgren and Johan Nordlander, for their guidance and patience in allowing me the time *to do it right*. I would also like to thank my friends and colleagues at EISLAB who have made this endeavor an enjoyable experience. Andrey Kruglyak deserves a special mention for an extensive proof-reading, feedback, and guidance in the finer arts of the English language, which has gone far beyond the call of duty.

The work presented in this thesis was funded by the I2-Microclimate project, initiated by CASTT (Center for Automotive System Technology and Testing) and CDT (Center for Distance-spanning Technology).

List of Abbreviations

CAN	Controller area network
CISC	Complex instruction set computing
CRO	Concurrent reactive object
EDF	Earliest deadline first
GC	Garbage collection
IP	Internet protocol
LED	Light-emitting diode
RISC	Reduced instruction set computing
SRP	Stack resource policy
TCP	Transmission control protocol
TT	TinyTimber

Contents

Part I

Chapter 1

Introduction

An embedded system is a computer system that is a part of a larger device with hardware and mechanical parts. Such a system often has limited resources (such as processing power, memory, and power) and it typically has to meet hard real-time requirements.

Today, embedded systems can be found almost everywhere, ranging from mobile phones to anti-lock braking systems in vehicles. While the increasing performance of hardware (e.g., processing power, memory) allows manufacturers to incorporate more functionality into their devices, this also increases the complexity of embedded software. In many cases, one software system is designed to perform multiple tasks. When these tasks are fairly independent of each other, it is often beneficial to execute them concurrently. To demonstrate the advantages of concurrent execution we consider a simple example.

Let a system react to two independent events (e.g., pressing a button and receiving data from the network). The first event triggers a reaction A, and the second triggers a reaction B. On a system with processing power C, processing of A takes 2 ms, and of B takes 100 ms. In addition to this, processing of A must be completed within 4 ms and of B within 400 ms, and the minimum inter-arrival time of the events is equal to the respective deadlines. In the case when the system does not support concurrent execution, we may need to execute both reactions within the span of 4 ms. This means that in the absence of concurrency, the requirements ($25.5C$) greatly exceed the available processing power C. On the other hand, if we can preempt B and execute A concurrently, we only need $0.75C$ of processing power. In other words, concurrency allows us to meet the requirements with much less processing power and to increase utilization of the system.

This example clearly demonstrates the benefits of introducing concurrency on a single-core system. In addition, in the pursuit of increased performance and lower power consumption, the trend in the embedded systems world is towards multi-core and multi-processor systems [1]. This inevitably introduces concurrency into the system, and with it concurrency-related problems; the most prominent is how to guarantee state consistency under concurrent modification.

The most common way to introduce concurrency is to use explicit threading supported

by a real-time operating system such as FreeRTOS [2] or OSE [3]. Then concurrency-related problems are addressed using synchronization primitives (mutexes, semaphores, etc.). The problem with this approach is that it is up to the programmer to ensure a correct and efficient use of these primitives. In "Software and the Concurrency Revolution" [4], a case against threads, shared state and locks is made by Herb Sutter and James Larus, and a similar argument is made in Edward Lee's paper "The problem with threads" [5]. The conclusion drawn in both papers is that we need a new way to express concurrency in order to be able to write correct concurrent code while retaining the ability to produce an efficient executable.

1.1 Research Questions

In this thesis we focus on concurrent reactive object model manifested in the reactive, object-oriented programming language Timber [6, 7] and in TinyTimber [8], a subset of the C language closely matching Timber's execution model. This implicit object-level concurrency model offers an abstraction on top of threads of execution, but as any abstraction it comes with a cost, which for resource-constrained embedded systems cannot be considered negligible. With this in mind, I have formulated the following research question:

Q1: *What is the cost of using concurrent reactive objects?*

The goal is to quantify the costs of using concurrent reactive objects in TinyTimber (such as code size, kernel data size, message passing overhead, scheduling overhead, context switching overhead) and to compare them to the costs of using the more common thread-based approach. Once we have quantified the costs, the next question arises:

Q2: *Is it possible to mitigate the cost of using concurrent reactive objects?*

This question will be answered by considering a specific implementation technique for TinyTimber based on earliest deadline first scheduling with stack resource policy (SRP). Our implementation is limited to SRP on a single core system. However, it should be possible to extend our implementation to the case of multi-core and multi-processor systems by utilizing existing work on Multiprocessor SRP [9].

For any hard real-time embedded systems, verification of correctness of system behavior (including its timing behavior and memory consumption) is often critical. Timber (but not TinyTimber) offers automatic memory management [10] which greatly simplifies writing programs; however, it also requires verification of timeliness of garbage collection if a correct behavior of the system should be guaranteed. Thus we need some kind of feasibility analysis for garbage collection. My next research question deals with this issue:

Q3: *How can feasibility of garbage collection be verified for a system based on the concurrent reactive object model?*

This comes down to verifying that there is enough time and memory in the system to perform garbage collection.

1.2 Structure of the Thesis

This thesis is divided into two parts. The first part includes introduction (chapter 1), background (chapter 2) and discussion of related work (chapter 3), followed by a summary of the papers included in this thesis (chapter 4) and conclusions (chapter 5). An important part of the background chapter is comprised by a description of the concurrent reactive object model. The second part of the thesis consists of three papers presented at conferences with a peer-review system, with my contribution to each paper clearly specified.

Chapter 2

Background

This chapter provides an essential background to the work presented in this thesis. We start by describing the concurrent reactive object model. Then we consider a number of execution time estimation and measurement techniques, introduce the stack resource policy and principles of garbage collection.

2.1 Concurrent Reactive Object Model

The concurrent reactive object (CRO) model is the execution and concurrency model of the Timber programming language, a general-purpose object-oriented language that primarily targets real-time systems [6, 7, 11]. A subset of C implementing the core features of Timber and also using the CRO model as its execution model is called TinyTimber [8]. TinyTimber has been used for implementation of a lightweight modular TCP/IP stack with support for "IP over CAN" and for teaching real-time programming (see my earlier work [12, 13]).

The costs of a TinyTimber implementation of the CRO model are considered in Paper A, and in Paper B this implementation is extended to support stack resource policy, which is used to decrease these costs. Paper C addresses a separate issue of verification of a real-time system with automatic memory management. The feasibility test for real-time garbage collection, presented in Paper C, is applicable to any CRO system. However, the presented experimental results were obtained for a Timber implementation, since TinyTimber targets systems with minimal resources and does not, in today's configuration, support automatic memory management.

In this section we briefly describe the main features of the CRO model. We focus on reactivity; object-orientation with a complete state encapsulation; object-level concurrency with message passing between objects; and the ability to specify timing behavior of a system.

Reactivity

Reactivity is the defining property of the CRO model, which makes it particularly suitable for embedded systems, since functionality of most, if not all, embedded systems can be expressed in terms of reactions to external stimuli and timer events. A reactive system can be described as follows: initially the system is idle, an external stimulus (originating in the system's environment) or a timer event triggers a burst of activity, and eventually the system returns to the idle state. A reactive object is either actively executing a method in response to an external stimulus or a message from another object, or passively maintains its state. Since initially the system is idle, some external stimulus is needed to trigger activity in the system.

Concurrent reactive objects can be used to model the system itself and its interaction with its environment (e.g., via sensors, buttons, keyboards, displays). Events in the physical world (such as pushing a button) result in an asynchronous message being sent to a handler object, and system output (e.g., flashing an LED) is represented as messages sent from an object to the environment.

Objects and state encapsulation

The CRO model specifies that all system state is encapsulated in objects. Each object has a number of methods, and the encapsulated state is only accessible from the object's methods. This is also known as a complete state encapsulation.

Methods of two objects can be executed concurrently, but each method is granted an exclusive access to its object's state, so only one method of an object can be active at any given time. Coupled with a complete state encapsulation, this provides a mechanism for guaranteeing state consistency under concurrent execution. The source of concurrency in a system can either be two external stimuli that are handled by different objects, or an asynchronous message sent from one object to another (more about message passing below).

To ensure that execution is reactive in its nature, each method must follow run-to-end semantics [14], i.e. it is not allowed to block execution waiting for an external stimulus or a message. An example of this would be an object representing a queue: if a *dequeue* method is invoked on an empty queue, it is not allowed to wait until data becomes available, but it must instead return a result indicating that the queue is empty.

Message passing and specification of timing behavior

In the CRO model objects communicate by passing messages. Each message specifies a recipient object and a method of this object that will be invoked. A message is either synchronous or asynchronous. The sender of a synchronous message blocks waiting for the invoked method to complete, while the sender of an asynchronous message can continue execution concurrently with the invoked method. Thus asynchronous messages introduce concurrency into the system. An asynchronous message can also be delayed by a certain amount of time.

Timing behavior of a system can be specified by defining a baseline and a deadline for an asynchronous message (a synchronous message always inherits the timing specification of the sender). The *baseline* specifies the earliest point in time when a message becomes eligible for execution, which for an external stimulus corresponds to its "arrival time" and for a message sent from one object to another is defined directly in the code. If the defined baseline is in the future, this corresponds to delaying the delivery of the message. The *deadline* specifies the latest point in time when a message must complete execution, which is always defined relative to the baseline.

Other Popular Concurrency Models

The most common concurrency model today is explicit concurrency, which is typically implemented using threads, e.g. POSIX *pthreads*. In the thread-based concurrency model, all threads can access the system state, and each thread is created explicitly. To be able to ensure state consistency under concurrent execution, a number of synchronization primitives are typically provided (such as semaphores, mutexes, etc.), but the correct behavior of the system relies on the programmer using these primitives correctly. Since each thread is created explicitly, the programmer is also tasked with manually dividing the system into concurrent parts. This is different from the approach taken in the CRO model, where both concurrency and state consistency under concurrent execution are implicit in the model.

Another concurrency model is the actor model [15]. Actors are independent concurrent entities that communicate using asynchronous messages. Upon receiving a message, an actor may execute its behavior, which may include sending a finite number of messages to other actors, creating a finite number of new actors, and designating a new behavior for the next message received. The most obvious difference from the CRO model is that the actor model lacks synchronous messages; however, the same behavior can normally be modeled using asynchronous messages. Another interesting difference is how actors behave compared to objects in the CRO model. An actor actively chooses which message to act upon and when, while an object must act upon all messages when they are delivered (one at a time), the actor is *active* while the object is *reactive*.

The concurrency model of TinyOS [16], the operating system that has gained a lot of traction in the area of low-power wireless sensor networks, features two levels of concurrency: *events* and *tasks*. Tasks always run to completion, i.e. they are not allowed to preempt other tasks. Events, on the other hand, are allowed to preempt tasks but sharing of state between tasks and events is not allowed. Both tasks and events are allowed to schedule other tasks. By default, all tasks are scheduled in first-in-first-out order, but a priority-based scheduling is possible by changing the scheduler. In comparison to the CRO model, the concurrency model of TinyOS is much more restrictive, as task scheduling in TinyOS can be expressed using CROs by defining all tasks as methods of a single object and by defining each event as a single method of a new object.

It is important to note that unlike the CRO model, neither of these alternative models natively incorporates a specification of timing behavior of a system.

2.2 Execution Time Estimation and Measurement

One of the costs of using the CRO model that we want to quantify is execution time of kernel operations such as posting a message or performing a context switch. In this section we describe two well-known approaches to quantifying this cost that we used in our work, namely execution time estimation using static code analysis and execution time measurement.

Execution Time Estimation Using Static Code Analysis

One of the goals of static code analysis is to estimate execution time of a given sequence of instructions (or a *program trace*). The simplest form of such analysis is counting the number of machine or assembly instructions. While this method is quite simple, its accuracy depends on the architecture of the processor. If the architecture is RISC-based, then there is a strong correlation between the number of instructions and the execution time. However, if the architecture is CISC-based, the correlation is weaker, since many CISC instructions require more than one cycle to execute, and in the worst case the number of cycles may depend on the operands of the instruction.

Most embedded systems today have a RISC-based instruction set with a limited number of CISC instructions. Depending on the characteristics of the code we wish to analyze, it might be enough to multiply the number of instructions with the average number of cycles per instruction. In some cases (e.g., when we have a small number of instructions) it would be better to weigh each instruction with the number of cycles required for execution.

In Paper A, we consider execution of TinyTimber kernel on PIC18 [17], AVR [18], ARM7 [19], and MSP430 [20]. The first three are RISC processors and we simply count the number of instructions; for MSP430, which is a CISC processor, we combine counting the number of instructions with measuring the actual execution time (see below). All program traces were manually extracted from the code.

Execution Time Measurement

As an alternative to execution time estimation using static code analysis, we can perform execution time measurement, which does not rely on any assumptions regarding behavior of the hardware. However, while a measured performance is closer to the actual performance, we need to take special care when we generate input for a program to observe the desired behavior (best-, average-, or worst-case execution time). This is a problem from the area of worst-case execution time analysis [21] and it is known as data-dependent control flow, i.e. the control flow (or execution path) of a program is dependent on the input data and hence so is the execution time. In general, there is no way to guarantee that the desired behavior is observed without testing all possible input combinations.

Measurement of execution time can be performed either on actual or simulated hardware. While simulated hardware in many cases allows for a more controlled execution with more information available, it is only an approximation of the actual hardware

(with possible simplifications such as omitted caches, branch prediction, multiple-issue capabilities, bus arbitration), resulting in a less accurate measurement.

A good overview of existing methods for measuring execution time on actual hardware is provided in [22]. These range from pure software to pure hardware methods. In general, hardware-assisted methods provide a much higher accuracy and granularity, but they usually require more effort (e.g, we might have to filter and analyze data), while pure software methods (e.g., the *time* command in UNIX) only require executing the program to obtain some performance metrics. Here we only discuss hardware-assisted methods since they were used in our work. The most accurate method is to use dedicated hardware (such as ARM ETM [23]) to monitor buses (e.g., instruction bus, data bus) and internal state of a processor (e.g., registers, processor flags) in real-time, since no modification of software is required and execution is only observed, not altered. With this method we can gather essentially *all* available information, but it also requires extensive data processing to identify relevant timing information. This method is similar to the method used in Paper B, where we used ARM Cortex-M3 with JTAG [24].

While all of the previously discussed methods require some external hardware, there are usually internal timers (or counters) that can be utilized to measure execution time. But since we do not have any means of transferring this information in real-time (if we do, we should use that for logging), we must store it in processor memory. This often increases the timing error and it also limits the number of events that can be logged since we have a finite amount of memory. Hence ensuring that the timing error is small and constant requires intimate knowledge of the processor architecture. This technique for measuring execution time was used in Paper C for an ARM7 processor.

2.3 Stack Resource Policy

Stack resource policy (SRP) is a policy for scheduling real-time tasks with shared resources that permits tasks with different priorities to share a single run-time stack [25]. SRP applies directly to scheduling policies with static and dynamic priority, including earliest deadline first (EDF), which is used by both Timber and TinyTimber kernels. It is one of the techniques we can use to decrease the cost of using concurrent reactive objects, as it decreases the number of context switches. It also simplifies the locking mechanism used to grant a method an exclusive access to its object's state.

The traditional version of SRP only addresses single-core systems, and our implementation (presented in Paper B) only applies to such systems. However, SRP has also been extended to multi-core and multi-processor systems (see, for example, [9] and [26]), and it should be possible to extend our implementation to support multi-core.

Here we briefly describe the main concepts of SRP, namely *jobs*, *resources*, and *resource ceilings*, and discuss how the CRO model can be mapped to SRP jobs and resources.

Jobs

SRP introduces the concept of *jobs*, defined as finite sequences of instructions with a known worst-case execution time, fixed resource usage, and priority. A job is executed in response to a *job request* that arrives at a certain point in time. Here $J_1, \ldots, J_n$ denote jobs and $\rho(J_i)$ denotes the *priority* of the job i. The priority of a job reflects its importance so that if $\rho(J_i) > \rho(J_j)$ then execution of J_j can be delayed in favor of executing J_i.

Resources

A system may have a number of non-preemptible (possibly multi-unit) resources $R_1, \ldots, R_n$. A *non-preemptible multi-unit resource* R_i is a resource with k units, and any number of units smaller than or equal to k can be requested, possibly by different jobs, but the total amount allocated at any time cannot exceed k.

A job may require zero or more of system resources. It acquires a certain number of units of a resource using a *request instruction* and the requested units are allocated to the job until it issues a *release instruction*. To avoid deadlock, each job must request and release resources in last-in-first-out order. In SRP, a job is only allowed to start execution when all resources it may need are available.

In order to allow for multiple jobs to share a single run-time stack, the stack is treated as a special non-preemptible resource. When a job is started, it is granted access to the stack and it may use it without explicitly requesting and releasing it. If a job J_i is preempted by another job J_j (which may only happen if $\rho(J_j) > \rho(J_i)$ and all resources that J_j may need are available), J_j is granted access to the part of the stack not currently used by J_i. This is possible since J_i would not resume execution until J_j has completed and released the stack.

Preemption levels

In addition to priority, a job is statically assigned a *preemption level* $\pi(J_i)$. A preemption level is related to the job's priority, but it is introduced as a separate property to support dynamic scheduling policies (such as earliest deadline first). Preemption levels can be defined in different ways, but the following condition must hold:

> C1: *If a job J_j can arrive after J_i and still have a higher priority than J_i, then $\pi(J_j)$ must be greater than $\pi(J_i)$.*

The definition of preemption levels for a system with static priorities is straightforward: $\pi(J_i)$ can be defined as equal to $\rho(J_i)$. However, for a system with dynamic priorities, the priority of a job is not statically known.

Let us assume that a job J_i has an *arrival time* $A(J_i)$ and a *relative deadline* $D(J_i)$. The arrival time varies between job requests while the relative deadline is constant. For earliest deadline first scheduling (used by Timber and TinyTimber implementations), the job's priority is defined so that $\rho(J_i) < \rho(J_j)$ if and only if $A(J_j)+D(J_j) < A(J_i)+D(J_i)$.

It is therefore consistent with (C1) to define the job's preemption level so that $\pi(J_i) < \pi(J_j)$ if and only if $D(J_i) > D(J_j)$, in other words, the preemption level of a job can be defined as *inversely proportional* to the relative deadline of the job.

Preemption ceilings

The concept of preemption ceilings in SRP is a refinement of the concept of *priority ceilings* [27] with priorities replaced by preemption levels. The values of preemption levels and preemption ceilings are both from the same ordered domain.

Each resource R_i has a *current ceiling* denoted by $\lceil R_i \rceil$. For multi-unit resources it depends on the number of available units and thus changes during execution, while for single-unit resources it is constant. Below we describe how a current ceiling of a resource can be defined.

Let us consider a job J_i and a resource R_j with current ceiling $\lceil R_j \rceil$. If J_i would need more units of R_j than are currently available, we should not let it preempt the currently executing job. This is achieved by ensuring that the current ceiling $\lceil R_j \rceil$ is greater than or equal to the preemption level $\pi(J_i)$. This must hold for any job that may request R_j and that is not currently executed.

If we restrict the SRP model to only support single-unit resources, which is sufficient to represent the CRO model, then the current ceiling of a resource can be statically defined as the maximum of zero and the preemption level of all jobs that may request the resource.

By introducing the concept of a *system ceiling*, equal to the maximum of current ceilings of all allocated resources and preemption levels of currently executing jobs, and only allowing a new job instance to begin execution when its preemption level is higher than the system ceiling, SRP ensures that jobs with higher priorities are executed first but only if all resources they may need are available. This guarantees that there will be no deadlock at run-time, and that any job instance with the highest priority will not wait for more than one job to release a required resource, which gives us a bounded priority inversion.

Translation of the CRO model

SRP offers a bounded priority inversion, deadlock free execution, sharing of run-time stack, and at most two context switches per job instance (since a job can never block waiting for a resource once started). Together with the shared run-time stack, this decreases the costs of using the CRO model.

In order to allow a CRO system to be represented in the SRP model, we must first translate the CRO model into jobs and resources. Assuming EDF scheduling, this translation is straightforward: each object is treated as a single-unit resource, each asynchronous message as a job (with an initial resource request for the destination object), and each synchronous message as a resource request. The baseline of a message corresponds to the arrival time of a job and the deadline to the relative deadline of the job.

2.4 Garbage Collection

In Paper C, we deal with schedulability analysis of garbage collection of real-time systems. This section provides an introduction to the main concepts of garbage collection.

Garbage collection is the process of automatically reclaiming unused memory, or *garbage*. There are two predominant approaches to garbage collection, namely *reference counting* and *root set scanning*. Reference counting is a local approach, i.e. for each allocated object a counter keeps track of the number of references to that object. This approach is simple to implement and is in its very nature *incremental* (algorithm is easily divided into small increments), which is normally considered advantageous for real-time systems. There is, however, a substantial performance penalty for keeping the reference count up-to-date during operations on the object. Reference counting also fails to properly analyze cyclic structures, such as when an object A holds a reference to object B and at the same time object B holds a reference to object A, which is not detected as garbage even if there are no other references to either A or B.

Root set scanning (a.k.a. tracing), on the other hand, is a global approach. In this approach, objects that are *reachable* from roots (i.e., there is a chain of references from a root to an object) are considered *live* (not garbage). Roots are typically addresses in CPU registers, global variables, and values on execution stack. How garbage is reclaimed depends on the specific algorithm, which can be *copying* [28], *non-copying* [29], and *hybrid* garbage collection [30]. Copying garbage collection relies on all heap memory divided into two parts, the *from-space* and the *to-space*. Allocations are normally performed in the *from*-space, until a garbage collection is triggered and then all live memory (objects reachable from the roots) are copied to the *to*-space, which subsequently becomes the *from*-space. Such garbage collection can either be performed incrementally (i.e., interleaved with execution of other tasks) or atomically (with execution of the tasks paused until garbage collection is complete). In the former case, allocations during garbage collection can be done either in the *from*-space or in the *to*-space, depending on the algorithm.

The problem with a non-copying garbage collector is that free memory becomes fragmented over time. This problem is usually solved by allocations of blocks of fixed size [31, 32] or by running defragmentation [33]. This problem is altogether avoided when using a copying garbage collector, such as the one presented in [28]. The main drawback of a copying collector, on the other hand, is that it requires twice as much memory, and there is also a cost for copying live memory. The main benefit, however, is that the garbage collection time is bound by the amount of live memory (as only reachable objects need to be copied).

In general, in a real-time system an incremental garbage collector is preferred, since the effect of collection on the execution of real-time tasks is significantly smaller. The collector used in Paper C is incremental and it is based on Cheney's copying garbage collector [34, 28].

Chapter 3

Related Work

In this chapter we describe the context of my work by looking at some related research. First we consider techniques for measuring code size and execution time (similar to those we use to measure kernel code size and overhead of using kernel primitives and program response times, respectively). Then we look into techniques for minimizing stack usage and preemption cost, which we try to minimize in order to decrease the cost of using concurrent reactive objects. Lastly, we consider different approaches to schedulability analysis of garbage collection.

Performance Measurements

Performance measurement of single-processor systems is a well-researched area. Since we have only used single processor systems, our measurement techniques are adaptations of well-known methods to our hardware platforms.

Measuring static code and data size is straightforward since most tool chains (such as GNU binutils [35], IAR Embedded Workbench [36], etc.) include utilities for obtaining these metrics. We also measure overhead of kernel primitives, such as sending a message, scheduling a message, context switching, etc. So let us now discuss a number of methods for measuring execution time that are similar to our techniques.

In [37], a tool for analyzing real-time embedded systems is presented. The recommended method for measuring performance is to instrument the code so that relevant state transitions are recorded by writing a value to an external I/O port of the processor. This port can be monitored with a logic analyzer to measure time between transitions. This technique is characterized by a high accuracy and a small impact of instrumenting the code on the program's execution time, and it was used in Paper A. In [38], a similar technique is used to compare two different real-time operating systems, as is done in Paper A where we compare FreeRTOS [2] and TinyTimber kernels.

In [39], the EMERALDS micro-kernel and its performance measurements are presented. The measurement technique is sampling an on-chip timer, which is similar to the method used in Paper B. The only difference is that we use built-in debugging hardware to sample the timer without modifying the code.

In [40], a family of garbage collection benchmarks for Real-Time Java is presented. Similarly to [39], a timer is sampled at different critical instants (e.g., at the start and end of garbage collection), but to minimize the impact of measurements on execution time, the sampled time is stored in a buffer for offline analysis after the test is complete. This is the same technique that we use in Paper C, though our method is adapted for use with an embedded system, i.e. we must transfer the data from the internal memory of the processor to a standard PC.

Cost Mitigation Techniques

In our work, we have used SRP to mitigate the cost of using the CRO model. The costs that we aim to decrease are code and data size of the kernel, message passing overhead, and context switching overhead. Here we consider a number of related techniques.

In [41], the authors present a technique in EMERALDS-OSEK micro-kernel that allows certain type of tasks, defined in the OSEK/VDX standard [42], to share a single stack. In Paper B, we present a similar technique; however, we allow all tasks to share a single stack.

In [43], Hofmeijer et al. present the EDFI scheduler of AmbientRT, a lightweight real-time earliest-deadline-first scheduler with deadline inheritance over shared resources. Their approach is similar to our EDF+SRP, as presented in Paper B. While our work is based on Baker's stack resource policy protocol [25], the EDFI scheduler is based on the scheduler from [44], which is a simplified version of SRP. They both lead to a reduced stack size and context switching cost.

Schedulability Analysis for Garbage Collection

Here we present related work addressing some issues related to garbage collection, such as determining the amount of additional work (i.e., execution time) for the garbage collector generated by a task, determining when garbage collection should be triggered, etc.

In [45], a garbage collection feasibility test based on response time analysis [46] is presented. Each task is assigned a cost in terms of garbage collection time per task invocation. This cost is defined as the maximum amount of work for the collector that a task invocation can generate, which is similar to formula (2) in Paper C. However, unlike in Paper C, no connection is made between the amount of work (i.e., the execution time of the collector) and the garbage collection algorithm.

This feasibility test is extended in [47] to allow for computation of an upper bound on the cycle time of the collector. The cycle time, i.e. the period between invocations of the garbage collector, directly affects memory requirements. This upper bound corresponds to the memory requirements specified in formula (4) in Paper C.

Another aspect of a garbage collection algorithm is when collection is triggered. This can be done either at regular intervals (time-triggered collection) or whenever the amount of allocated memory crosses a certain threshold (work-triggered collection). In [48], the authors compare these two approaches and demonstrate that time-triggered collection

is superior in terms of predictability of garbage collection overhead. This validates the choice of time-triggered garbage collection in Paper C.

Time-triggered garbage collection can be periodic, when the collector runs with the highest priority, or slack-based, when it is assigned the lowest priority of all the tasks in the system. These two approaches to scheduling garbage collector are evaluated in [30]. It is demonstrated that they have distinct limitations, e.g. a particular system might require slack-based collection to be feasible while another might require a periodic collection. Thus the choice of scheduling policy is a key part of designing garbage collection for a particular real-time system. We note that the feasibility analysis presented in Paper C is only valid for slack-based scheduling.

Chapter 4

Summary of Papers

In this chapter we give a summary of the papers included in Part II of this thesis. In Paper A, we describe TinyTimber [8], a C implementation of the concurrent reactive object (CRO) model, present performance measurements for the TinyTimber kernel on a number of hardware platforms and compare it to the FreeRTOS kernel [2]. In Paper B, we develop a unified scheduling of reactions to external events (originating in the system's environment) and internal events (originating within the system) based on earliest deadline first (EDF) scheduling with stack resource policy (SRP). Paper C introduces a feasibility test for scheduling garbage collection in a reactive real-time system.

4.1 Summary of Paper A

In this paper we describe the concurrent reactive object model of Timber [6] and its implementation in C (TinyTimber, TT [8]). Performance measurements for the primitive operations of TT kernel, such as sending a synchronous or an asynchronous message, are presented for a number of hardware platforms: PIC18 [17], AVR [18], MSP430 [20], and ARM7 [19]. The technique used to estimate execution time is static code analysis (see chapter 2). The lower-end PIC18 platform requires the largest number of instructions, followed by the AVR, while MSP430 and ARM7 require the fewest.

A comparison of two different implementations of an application is also presented. The first uses thread-based concurrency model, as implemented by FreeRTOS, and the second uses the CRO model, as implemented by TT. The comparison is made on MSP430 and the technique used for timing measurements is monitoring of I/O ports of the processor with an oscilloscope, as described in chapter 2.

Three distinct timing metrics are used in this paper. The first metric is the time for handling an external event, that is the time between an external event and completion of processing it. The results are in favor of TT: 150μs vs. 220μs. The second metric is the jitter in pulse length. Here the result is clearly in favor of TT: 20μs vs. 1ms. The last metric is the longest period of time during which external events are blocked. The result is yet again in favor of TT: 100μs vs. 120μs.

The memory footprint of the application (including the kernel) is also presented in the paper and is, in terms of both code and data size, in favor of TT: code size 3544 vs. 5304 bytes and data size 1178 vs. 1928 bytes.

4.2 Summary of Paper B

An embedded system can normally be modeled in terms of time-constrained reactions to events. These events are either internal, originating within the system, or external, originating from the system's environment. In general, a reaction to an internal event is scheduled by the kernel, while a reaction to an external event is scheduled by a hardware scheduler (i.e., interrupt hardware). The most common approach to designing a real-time operating system is to expose this non-uniform scheduling to the programmer and treat these reactions differently. Reactions to external events are typically given precedence over reactions to internal events and in order to share resources between reactions to internal and external events, the hardware scheduler must be explicitly prevented from running (e.g., by disabling or masking interrupts). This complicates not only the design, but also analysis of the system. In this paper we propose a uniform system view where reactions to both external an internal events are scheduled uniformly by the kernel. This is done using an earliest deadline first (EDF) scheduler with stack resource policy (SRP). A TinyTimber-based implementation of this scheduler for a Cortex-M3 [49] is also presented.

A formal verification of an EDF+SRP system can be performed using the *enhanced processor-demand test* described by Baruah in [50]. This test requires that the timing properties of the tasks (such as inter-arrival times, wost-case execution times, deadlines, etc.) are known. However, it does not explicitly account for the overhead of scheduling tasks and resource synchronization (requesting and granting of resources), instead this overhead is typically added to the worst-case execution time of each task and thus affects the schedulability of the system. In our implementation, scheduling a task corresponds to handling of an event and resource synchronization to a synchronous call. We measure this overhead using several timing metrics (obtained using JTAG hardware, see chapter 2):

- The first metric is the delay of an external event, that is the time between release of a task triggered from outside the system and execution of the first instruction of the task. This delay is at least 123 clock cycles (1.23 μs at 100MHz).
- The second metric is the delay of an internal event, that is the time between release of a task triggered from within the system and execution of the first instruction of the task. This delay is at least 138 clock cycles (1.38 μs at 100MHz).
- The third timing metric is the cost of performing a synchronous call (requesting a resource and invoking a method), that is the time between invoking a synchronous call and execution of the first instruction of the method, which is always 31 clock cycles (0.31 μs at 100MHz).

It should be noted that the delay of both internal and external events depends on the current length of the task queue. The results above are for the best case (an empty queue), and the maximum length of the queue affects the tightness (accuracy) of the feasibility test. To show how this length affects handling of events, the time of performing a queue operation when handling an internal event is measured; as the queue length changes from zero to four, this operation takes between 127 and 172 clock cycles (1.27 and 1.72 μs at 100MHz).

The memory footprint of the EDF+SRP implementation of the kernel is 644 bytes for code and 20 bytes for data. The memory overhead of using this implementation of EDF+SRP is 24 bytes for each task and 4 bytes for each resource.

4.3 Summary of Paper C

The focus of this paper is schedulability of garbage collection in a real-time system with automatic memory management. In order to verify correctness of such a system, we must ensure that all tasks meet their deadlines and that the system does not run out of memory. There are a number of feasibility test that can be used to verify that a set of tasks will meet their deadlines, such as *response time analysis* [46] or *processor demand analysis* [51], but these tests do not incorporate the cost of garbage collection. The common approach is to extend the analysis by incorporating the garbage collector as just another task. Our approach, on the other hand, is to decouple the cost of garbage collection from the schedulability analysis.

To decouple the cost of garbage collection, an incremental collector is used which is only allowed to run if no other task is runnable (this is known as idle time garbage collection). This requires that the system becomes idle at some point, which mandates a reactive approach to system design.

Under the assumption that the CRO execution model is used, we formulate the *garbage demand analysis*. This analysis models the demand of the collector during a period of time t (formula (3) in Paper C) in terms of worst-case execution time of collector operations (such as scanning an address, copying a word of data, etc.). It also models memory requirements (formula (4) in Paper C) for the system with a time-triggered, slack-based (idle time) garbage collector with a period t.

The analysis assumes that for a specific hardware platform, the worst-case execution time of each collector operation is constant. To validate this assumption, five different applications were tested, each written to isolate a specific term in formula (4) in Paper C (e.g., reachable memory, number of reachable nodes, and number of reachable references). To gather the necessary information during the test, the collector implemented in the Timber run-time system was modified to perform logging of transitions (i.e., start and completion of garbage collection). The results vindicate our approach, in particular the use of constants for worst-case execution times of each collector operation.

Chapter 5

Conclusions

Let us now demonstrate how the results of the papers provide answers to the research questions.

Q1: *What is the cost of using concurrent reactive objects?*

In Paper A we quantify the overhead of using concurrent reactive objects on a number of hardware platforms. The overhead is measured in terms of the number of instructions required for message passing and context switching. In addition, we compare the performance of TinyTimber and FreeRTOS kernel on MSP430, the former representing the CRO model and the latter a thread-based concurrency model. The comparison establishes significant similarities in the level of performance offered by the two implementations, but the CRO model is shown to have a clear advantage with respect to limiting the jitter in output pulse length.

Q2: *Is it possible to mitigate the cost of using concurrent reactive objects?*

In Paper B, we demonstrate how earliest-deadline-first scheduling with stack resource policy can be used to schedule a reactive system. Stack sharing in SRP decreases memory requirements of a system, and the limitation of the number of context switches (at most two per task) is clearly advantageous for performance. In addition, we demonstrate that the cost of a synchronous call is only 31 clock cycles, which can be compared to the measurements for a non-SRP implementation presented in Paper A, where the cost of a synchronous call is 50 instructions and hence at least 50 clock cycles.

Q3: *How can feasibility of garbage collection be verified for a system based on the concurrent reactive object model?*

The analysis developed in Paper C can be used to verify that a CRO-based system has enough time to perform garbage collection, and to determine the system's memory requirements.

References

[1] C. H. K. van Berkel, "Multi-core for mobile phones," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 1260–1265. [Online]. Available: http://portal.acm.org/citation.cfm?id=1874620.1874924

[2] FreeRTOS-A Free RTOS for ARM7, ARM9, Cortex-M3, MSP430, MicroBlaze, Mar. 2011. [Online]. Available: http://www.freeRTOS.org

[3] Enea OSE: Multicore Real-Time Operating System, Mar. 2011. [Online]. Available: http://www.enea.com/ose

[4] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, pp. 54–62, September 2005. [Online]. Available: http://doi.acm.org/10.1145/1095408.1095421

[5] E. A. Lee, "The problem with threads," *Computer*, vol. 39, pp. 33–42, May 2006. [Online]. Available: http://portal.acm.org/citation.cfm?id=1137232.1137289

[6] M. Carlsson, J. Nordlander, and D. Kieburtz, "The semantic layers of timber." in *APLAS'03*, 2003, pp. 339–356.

[7] B. Andrew P., C. Magnus, J. Mark P., K. Richard, and N. Johan, "Timber: A programming language for real-time embedded systems," Tech. Rep., 2002.

[8] J. Eriksson, "Embedded real-time software using TinyTimber : reactive objects in C," p. 84, 2007. [Online]. Available: http://epubl.ltu.se/1402-1757/2007/72/LTU-LIC-0772-SE.pdf

[9] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, Dec. 2001, pp. 73 – 83.

[10] P. R. Wilson, "Uniprocessor garbage collection techniques," in *Proceedings of the International Workshop on Memory Management*, ser. IWMM '92. London, UK: Springer-Verlag, 1992, pp. 1–42. [Online]. Available: http://portal.acm.org/citation.cfm?id=645648.664824

[11] The Timber Language, Mar. 2011. [Online]. Available: http://timber-lang.org

[12] P. Lindgren, S. Aittamaa, and J. Eriksson, "IP over CAN, Transparent Vehicular to Infrastructure Access," in *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, Jan. 2008, pp. 758 –759.

[13] P. Lindgren, J. Nordlander, K. Hyyppä, S. Aittamaa, and J. Eriksson, "Comprehensive Reactive Real-Time Programming," in *Hawaii International Conference on Education: 2008 Conference Proceedings*, May 2008, pp. 1440–1448.

[14] P. Lindgren, J. Nordlander, L. Svensson, and J. Eriksson, "Time for Timber," Luleå University of Technology, Tech. Rep., 2005. [Online]. Available: http://pure.ltu.se/ws/fbspretrieve/299960

[15] G. Agha, *Actors: a model of concurrent computation in distributed systems.* Cambridge, MA, USA: MIT Press, 1986.

[16] TinyOS: Operating System for Low-power Wireless Sensor Nodes, Mar. 2011. [Online]. Available: http://www.tinyos.net

[17] Microchip PIC18 Processor, Mar. 2011. [Online]. Available: http://www.microchip.com/en_US/family/8bit/

[18] Atmel AVR, Mar. 2011. [Online]. Available: http://www.atmel.com/avr

[19] ARM7 Processor Family, Mar. 2011. [Online]. Available: http://www.arm.com/products/processors/classic/arm7/index.php

[20] Texas Instrument MSP430, Mar. 2011. [Online]. Available: http://www.ti.com

[21] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. B. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution-time problem - overview of methods and survey of tools," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 3, 2008.

[22] D. B. Stewart, "Measuring execution time and real-time performance," in *In: Proceedings of the Embedded Systems Conference (ESC SF)*, 2002, pp. 1–15.

[23] ARM Coresight Trace Macrocells, Mar. 2011. [Online]. Available: http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/index.php

[24] ARM Coresight for Cortex-M Series Processor, Mar. 2011. [Online]. Available: http://www.arm.com/products/system-ip/debug-trace/coresight-for-cortex-m.php

[25] T. Baker, "A stack-based resource allocation policy for realtime processes," in *Real-Time Systems Symposium, 1990. Proceedings., 11th*, Dec. 1990, pp. 191 –200.

[26] P. Gai, M. Di Natale, G. Lipari, A. Ferrari, C. Gabellini, and P. Marceca, "A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform," in *Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. The 9th IEEE*, May 2003, pp. 189 – 198.

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Trans. Comput.*, vol. 39, pp. 1175–1185, September 1990. [Online]. Available: http://dx.doi.org/10.1109/12.57058

[28] C. J. Cheney, "A nonrecursive list compacting algorithm," *Commun. ACM*, vol. 13, pp. 677–678, November 1970. [Online]. Available: http://doi.acm.org/10.1145/362790.362798

[29] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Commun. ACM*, vol. 3, pp. 184–195, April 1960. [Online]. Available: http://doi.acm.org/10.1145/367177.367199

[30] T. Kalibera, F. Pizlo, A. Hosking, and J. Vitek, "Scheduling hard real-time garbage collection," in *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, Dec. 2009, pp. 81 –92.

[31] W. T. Comfort, "Multiword list items," *Commun. ACM*, vol. 7, pp. 357–362, June 1964. [Online]. Available: http://doi.acm.org/10.1145/512274.512288

[32] K. C. Knowlton, "A fast storage allocator," *Commun. ACM*, vol. 8, pp. 623–624, October 1965. [Online]. Available: http://doi.acm.org/10.1145/365628.365655

[33] Berkeley, Edmund Callis, and Bobrow, Daniel G. and Information International, Inc. , *The programming language LISP; its operation and applications. Editors: Edmund C. Berkeley and Daniel G. Bobrow.* M.I.T. Press Cambridge, Mass.,, 1966.

[34] M. Kero, J. Nordlander, and P. Lindgren, "A correct and useful incremental copying garbage collector," in *Proceedings of the 6th international symposium on Memory management*, ser. ISMM '07. New York, NY, USA: ACM, 2007, pp. 129–140. [Online]. Available: http://doi.acm.org/10.1145/1296907.1296924

[35] GNU Binutils, Mar. 2011. [Online]. Available: http://www.gnu.org/software/binutils/

[36] IAR Embedded Workbench, Mar. 2011. [Online]. Available: http://www.iar.com

[37] D. Stewart and G. Arora, "A tool for analyzing and fine tuning the real-time properties of an embedded system," *Software Engineering, IEEE Transactions on*, vol. 29, no. 4, pp. 311 – 326, Apr. 2003.

[38] K. Weiss, T. Steckstor, and W. Rosenstiel, "Performance analysis of a rtos by emulation of an embedded system," in *Proceedings of the Tenth IEEE*

International Workshop on Rapid System Prototyping, ser. RSP '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 146–. [Online]. Available: http://portal.acm.org/citation.cfm?id=519625.828165

[39] K. Zuberi and K. Shin, "Emeralds: a microkernel for embedded real-time systems," *Real-Time and Embedded Technology and Applications Symposium, IEEE*, vol. 0, p. 241, 1996.

[40] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, and J. Vitek, "Cdx: A family of real-time java benchmarks," 2009.

[41] K. Z. Padmanabhan, P. Pillai, and K. G. Shin, "Emeralds-osek: A small real-time operating system for automotive control and monitoring," in *n Proc. SAE International Congress & Exhibiton*, 1999.

[42] OSEK - Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug, Mar. 2011. [Online]. Available: http://www.osek-vdx.org/

[43] T. Hofmeijer, S. Dulman, P. Jansen, and P. Havinga, "Ambientrt - real time system software support for data centric sensor networks," in *Intelligent Sensors, Sensor Networks and Information Processing Conference, 2004. Proceedings of the 2004*, Dec. 2004, pp. 61 – 66.

[44] P. G. Jansen, S. J. Mullender, P. J. Havinga, and H. Scholten, "Lightweight edf scheduling with deadline inheritance," 2003. [Online]. Available: http://doc.utwente.nl/41399/

[45] R. Henriksson and R. Henriksson, "Predictable automatic memory management for embedded systems," in *In Proc. of the Workshop on Garbage Collection and Memory Management. ACM SIGPLAN-SIGACT*, 1997.

[46] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal*, vol. 29, no. 5, pp. 390–395, May 1986. [Online]. Available: http://dx.doi.org/10.1093/comjnl/29.5.390

[47] S. G. Robertz and R. Henriksson, "Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems," *SIGPLAN Not.*, vol. 38, pp. 93–102, June 2003. [Online]. Available: http://doi.acm.org/10.1145/780731.780745

[48] D. F. Bacon, P. Cheng, and V. T. Rajan, "A real-time garbage collector with low overhead and consistent utilization," *SIGPLAN Not.*, vol. 38, pp. 285–298, January 2003. [Online]. Available: http://doi.acm.org/10.1145/640128.604155

[49] ARM Cortex-M3, Mar. 2011. [Online]. Available: http://www.arm.com/products/processors/cortex-m/cortex-m3.php

[50] S. K. Baruah, "Resource sharing in edf-scheduled systems: A closer look," in *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, Dec. 2006, pp. 379–387.

[51] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, pp. 301–324, 1990, 10.1007/BF01995675. [Online]. Available: http://dx.doi.org/10.1007/BF01995675

Part II

Paper A

TinyTimber, Reactive Objects in C for Real-Time Embedded Systems

Authors:
Per Lindgren, Johan Eriksson, Simon Aittamaa, and Johan Norlander

Contribution:
My contribution to this paper includes implementing the example applications, modifying TinyTimber and FreeRTOS kernels to allow for performance measurements, discussion and analysis of the results, and writing the description of the TinyTimber kernel and the application interface.

Reformatted version of paper accepted for publication in:
Design, Automation, and Test in Europe (DATE) 2008

TinyTimber, Reactive Objects in C for Real-Time Embedded Systems

Per Lindgren, Johan Eriksson, Simon, Aittamaa, Johan Nordlander

Abstract

Embedded systems are often operating under hard real-time constraints. Such systems are naturally described as time-bound reactions to external events, a point of view made manifest in the high-level programming and systems modeling language Timber. In this paper we demonstrate how the Timber semantics for parallel reactive objects translates to embedded real-time programming in C. This is accomplished through the use of a minimalistic Timber Run-Time system, TinyTimber (TT). The TT kernel ensures state integrity, and performs scheduling of events based on given time-bounds in compliance with the Timber semantics. In this way, we avoid the volatile task of explicitly coding parallelism in terms of processes/threads/semaphores/monitors, and side-step the delicate task to encode time-bounds into priorities.

In this paper, the TT kernel design is presented and performance metrics are presented for a number of representative embedded platforms, ranging from small 8-bit to more potent 32-bit micro controllers. The resulting system runs on bare metal, completely free of references to external code (even C-lib) which provides a solid basis for further analysis. In comparison to a traditional thread based real-time operating system for embedded applications (FreeRTOS), TT has tighter timing performance and considerably lower code complexity. In conclusion, TinyTimber is a viable alternative for implementing embedded real-time applications in C today.

1 Introduction

The ever increasing complexity of embedded systems operating under hard real-time constraints, sets new demands on rigorous system design and validation methodologies. Furthermore, scheduling for real-time embedded systems is known to be very challenging, mainly because the lack of tools that are able to extract the necessary scheduling information from the specification at different levels of abstraction [12]. However, in many cases, such embedded systems are naturally described as (chains of) time-bound reactions to external events, a view supported natively in the high-level programming and systems modeling language Timber in the form of reactive objects. These time bounds can be used directly as basis for both offline system analysis and during run-time scheduling. In contrast to *synchronous* reactive objects [8, 7] and synchronous languages [5], Timber inherently captures sporadic events, thus provides a more general approach to

reactive system modelling. The engineering perspectives of the Timber design paradigm are further elaborated in [14, 11, 13].

In this paper, we present *TinyTimber* (TT) - a minimalistic, portable real-time kernel with predictable memory and timing behavior - and we demonstrate how the Timber semantics translates to embedded real-time programming in C. Through the TT implementation, developers are provided a C interface to a minimalistic Timber Run-Time system, allowing C code to be executed under the reactive design paradigm of Timber (section 2). The TT kernel features the following subset of Timber;

- Concurrent, state protected objects
- Synchronous and asynchronous messages
- Deadline scheduling

In the design of the TT kernel, utmost care has been taken in order to offer bounded memory and timing behavior that is controllable by compile time parameters. The kernel itself is minimalistic, consisting solely of an event queue manager together with a real-time scheduler, and does neither rely on dynamic heap based memory accesses nor additional libraries. Thus, the functionality of the kernel can be made free of dependencies on third party code (even C-lib if so wished), which in turn benefits portability and robustness. Furthermore, the core functionality of the kernel is implemented in ANSI-C.

In the context of other minimalistic operating systems and kernels such as TinyOS [3], Contiki [10], FreeRTOS [2], and AmbientRT [1], TT stands out with its deadline-driven scheduling and the heritage to the reactive object paradigm of Timber. While TinyOS and Contiki lack native real-time support, FreeRTOS provides pre-emptive scheduling based on task priorities in a traditional fashion, and AmbientRT undertakes dynamic task scheduling based on most urgent deadlines, similar to our approach. However, TT differs fundamentally to AmbientRT in terms of our object-based locking mechanism, which allows true parallelism and hence may improve on schedulability and scalability to SRP like approaches [4]. Furthermore, TT provides best effort EDF scheduling with online resource management.

Based on experimental measurements carried out on a set of representative embedded platforms (PIC18, AVR5, MSP430 and ARM7), we show that the TT kernel can implement Timber semantics with high timing accuracy and low memory overhead. Furthermore, we compare the TT kernel to a thread based real-time operating system for embedded applications (FreeRTOS), and our experimental results verify that TT provides tighter timing performance, while matching resource requirements and having considerably lower code complexity. In conclusion, this paper shows that TinyTimber is a viable alternative for implementing real-time applications in C on embedded platforms.

2 Reactive Objects in Timber

In this section we briefly overview Timber in order to introduce the concepts that are most relevant to the rest of the paper. For an in depth description, we refer to the draft

language report [6], the formal semantics definition [9], and previous work on reactive objects [15, 16] and functional languages [17].

Timber seamlessly integrates the following concepts; concurrent objects with state protection, deadline scheduling of synchronous and asynchronous messages, higher-order functions, referential transparency, automatic memory management, and static type safety, with subtyping, parametric polymorphism and overloading.

However, it is the notion of *reactivity* that gives Timber its characteristic flavor. In effect: Timber methods never *block* for events, they are *invoked* by them. Timber unifies concurrent and object-orientated paradigms by its concept of concurrent state-protected objects (resources). The execution model of Timber ensures mutual exclusion between the methods of an object instance. This way Timber conveniently captures the inherent parallelism of a system without burdening the programmer with the volatile task of explicitly coding up parallelism in terms of traditional processes/threads/semaphores/-monitors, etc [14]. Designed with real-time applications in mind, the language provides a notion of timed reactions that associate each event with an absolute time-window for execution. Events are either generated by the environment (typically as interrupts, as in the case of software realizations of Timber models) or through synchronous/asynchronous message sends expressed directly in the language (not as OS primitives).

The Timber run-time model utilizes deadline scheduling directly on basis of programmer declared event information, which avoids the problem of turning deadlines into relative process priorities. In short:

- *Objects and parallelism:* The parallel and object oriented models go hand in hand. An object instance executes in parallel with the rest of the system, while the state encapsulated in the object is protected by forcing the methods of the object instance to execute under mutual exclusion. This implicit coding of parallelism and state integrity coincides with the intuition of a reactive object. Furthermore, all methods in a Timber program are non-blocking, hence a Timber system will never lack responsiveness due to intricate dependencies on events that are yet to occur.
- *Events, methods and time:* The semantics of Timber conceptually unifies events and methods in such a way that the specified constraints on the timely reaction to an event can be directly reused as run-time parameters for scheduling the corresponding method. Event *baseline* (release) and *deadline* define the permissible execution window for the corresponding method. All other points in time will be given relative to the event baseline, and will thus be free from jitter induced by the actual scheduling of methods.

3 Reactive Objects in C using TinyTimber

Over the last decades, C has become the dominating language for programming embedded systems. As the C language lacks native real-time support, concurrency and timing constraints are traditionally implemented through the use of external libraries, executing under some real-time operating system or kernel.

3.1 The TinyTimber application interface

TinyTimber (TT) allows C code to be executed under the pure reactive design paradigm of Timber. TT offers concurrent reactive objects (`Object`) with state protection as well as synchronous and asynchronous messages under deadline scheduling. Each message/event has a permissible interval for method execution (between the absolute points in time *baseline* and *deadline*). In case of synchronous messages (`SYNC(o, m, p)`), base- and deadlines are always inherited from the sender, (object `o`, method `m`, and parameter `p`). For asynchronous messages (`ASYNC(-1/b, -1/d, o, m, p)`), the new baseline can be either inherited (`-1`) or being computed as current *baseline* + `b`. However, if this new *baseline* (absolute point in time) has already passed at the time of posting, the new message *baseline* is set to current time. Respectively, the message *deadline* can be either inherited (`-1`), or set relative to the the new *baseline*. Events from external sources i.e. interrupts are (conceptually) executed with *baseline* and *deadline* set to current time.

Like its higher-level counterpart, TT assumes a run-to-end method semantic, hence no means are offered for synchronization with events by actively postponing execution within a method.

On the top level, a TT application consists of the implicit root object and its methods, i.e., the functions installed for handling interrupts and the reset signal. The state of the root object is the state of the globally declared C variables. To impose more structure, the root state can be further partitioned into objects of their own, whose methods are run under supervision by the scheduler. Method calls crossing object boundaries must never bypass the kernel primitives, otherwise any communication pattern between the objects can be devised.

A TT application is compiled and linked with the TT kernel for a target architecture using a C compiler suite such as gcc. For a bare-chip target, the executable image will not rely on additional libraries, although libraries may be incorporated as long as they do not violate the run-to-end assumption of TT.

4 The TinyTimber kernel

The primary job of the TT kernel is to manage the messages queues and schedule messages onto execution threads such that (if possible) all message deadlines are met. The TT kernel is purely event driven, hence, if there are no messages eligible for scheduling, the system is idle. The idle state may be used to put the system in low power mode. The complete source code of TT can be obtained from the authors under an open-source license.

4.1 Informal description

In the following we will discuss how the TT kernel addresses event scheduling, with respect to safety, liveness, and real-time properties [12].

These criteria are met by the TT kernel by Earliest Deadline First (EDF) scheduling with priority inheritance, together with mutual exclusion between object instance methods. Priority inheritance together with the run-to-end Timber semantics ensures that priority inversion will be bounded. As object instance methods operate under mutual exclusion the only possible source of deadlock is circular synchronous events. During runtime, the TT kernel will report such hazards. Applications free of such circular references will be executed in a *safe* (with respect to deadlock) manner by the TT kernel. Since the kernel is internally free of blocking operations, and given the run-to-end requirement on object methods, each event will eventually be executed and *liveness* of the system is upheld. This is achieved as the kernel itself will never enter the idle state unless no events are eligible for execution, and each event will eventually become released (as there is no notion of infinite baseline in Timber). The EDF scheduling together with the bounded priority inversion provides best effort *real-time* properties.

5 Performance Measurements

In this section we characterize the current TT implementation v0.3.0 on a number of representative platforms and give a brief comparison to a traditional thread-based open source operating system (FreeRTOS).

5.1 TinyTimber Overhead

Using TT there are two means of passing messages between objects, either synchronous or asynchronous. In the case of a synchronous message the TT kernel will; acquire the object lock (mutex), call the method specified, and release the object lock. The overhead cost of performing a synchronous call to an unlocked object is shown in Table 1 (a). However, if the object is locked the kernel will force the method holding the lock to resume execution (run-to-end) with inherited priority. The priority inversion is bounded by the acyclic chain of synchronous messages needed to be completed. Priority inversion overhead stems from implicit context swithces, (c) and (f).

In the case of an asynchronous message a baseline and a deadline is assigned to a synchronous message, effectively delaying the execution of the message (b). Once the baseline of the message expires a timer interrupt is generated, the context of the current tread will be saved (c), the message will be released into the queue of active messages (d), if the released message has the earliest deadline a new thread will be allocated (e), the context of the current thread will be restored (f), and a synchronous call is performed (a). Note that the cost of releasing the messages into the active queue is dependent on the number of messages to be released and the number of messages in the active queue, Table 1 (d) shows the cost when one message is released into an empty active queue (best case).

To preserve the state integrity of the kernel interrupts are disabled/enabled upon kernel entry/exit. The instruction count for the largest critical section (interrupts disabled) arises when a baseline(s) expires and message(s) is/are released into the active queue (g).

	PIC18	AVR5	MSP430	ARM7
(a) Synchronous Call	150	63	46	50
(b) Asynchronous Call	406	167	97	74
(c) Save Context	$137 + 7n$	56	18	23
(d) Release Message	228	53	50	37
(e) Allocate Thread	19	14	5	8
(f) Restore Context	$136 + 10n$	50	18	15
(g) Critical Section	884	398	213	224

Table 1: TinyTimber instruction count for a set of kernel mechanisms. All instruction counts are best case for current implementation. For PIC18, n gives the call-stack depth.

The critical section directly affects timing accuracy and responsiveness (with respect to external events).

5.2 Example Application

The desired behavior of the application is as follows; upon some external event a given output should be driven high for three milliseconds. Yet simple, the application exercises mechanisms for context switching, synchronization, and timing.

5.3 Application Implementation

The TT implementation is shown in Listing 1. The `output` object is created to encapsulate the output (pin 6.7). When the external interrupt is triggered an asynchronous message is sent to the `output` object invoking the `output_high` method. The `output_high` method will drive the output high and post an asynchronous message to the current object instance that will invoke the `output_low` method after three milliseconds.

The FreeRTOS implementation is shown in Listing 2. A single thread and semaphore is used to implement the desired behavior. Once the semaphore is released we drive the output high, delay for three milliseconds and drive the output low.

5.4 Application Comparison

To measure the response time the first channel of an oscilloscope was connected to the output, the second channel to the interrupt status, and the trigger to the external event. All measurements were performed on an idle system i.e., no other other threads/messages were running. The platform used for the measurements was a MSP430 (msp430x1611) clocked at 4.7MHz.

The delay is defined as the time between the triggering of the external event and the output driven high. The jitter is defined with respect to the pulse length, and the critical section is defined as the longest period of time interrupts are disabled. For the memory

	TinyTimber	FreeRTOS
Flash	3544 bytes	5304 bytes
RAM	1178 bytes	1928 bytes

Table 2: Memory footprints of example applications.

	TinyTimber	FreeRTOS
Delay	$150\mu s$	$220\mu s$
Jitter	$20\mu s$	$1ms$
Critical Section	$100\mu s$	$120\mu s$

Table 3: Delay, Jitter, and Critical Section length.

footprint of the application, flash is defined as the size of the `.text` section and RAM as the size of the `.data` and `.bss` sections.

Table 2 shows that the footprint of TT is significantly smaller, mainly due to its minimalistic API and simple implementation.

Table 3 shows that TT has a slight edge with respect to both delay and critical section. However, when it comes to timing accuracy, jitter measurements are clearly in favor of the TT implementation. FreeRTOS undertakes a system tick based timing scheme while TT uses a free running timer. The resolution of the system timer in FreeRTOS is 1kHz (by default) while the TT timer has a resolution of 32.768kHz (default on MSP430). The CPU overhead of a system tick based scheme is directly proportional to the timing resolution (where 1 kHz is a reasonable tradeoff). In the case of TT with its free running timer, the resolution is approx 30 μs (1/32768), with a measured average of 20 μs. For a system under load, the limiting factor for TT timing will be the critical section (approximately 100 μs). Hence, in practice a 10 kHz timer would be sufficient and TT could be expected to offer a tenfold improvement over FreeRTOS timing accuracy.

Listing 1: tt.c

```
#define PULSE_WIDTH MSEC(3)

#define initOutput(width, jitter) {initObject(), width}
typedef struct output_t {
    Object obj;
    Time width;
} output_t;
static output_t output = initOutput(PULSE_WIDTH);

env_result_t output_low(output_t *self, int arg) {
    P6OUT &= ~0x80; /* Set pin 6.7 low. */
}

env_result_t output_high(output_t *self, int arg) {
    P6OUT |= 0x80; /* Set pin 6.7 high. */
    ASYNC(self->width, 0, self, output_low, 0);
}

void msp430_port1_vector(void) {
    P1IFG = 0x00;   /* Clear intrrupt flag for pin 1.7. */
    ASYNC(-1, -1, &output, output_high, 0);
}

INTERRUPT(PORT1_VECTOR, msp430_port1_vector);

static void init(void) {
```

```
    P1SEL   &= ~0x80; /* Configure pin 1.7 as digital I/O. */
    P1DIR   &= ~0x80; /* Configure pin 1.7 as input. */
    P1IE    |= 0x80;  /* Enable interrupts for pin 1.7. */
    P6DIR   = 0x80;   /* Configure pin 6.7 as output. */
}
STARTUP(init);
```

Listing 2: freertos.c

```
#define PULSE_WIDTH ((3*configTICK_RATE_HZ+999)/1000)

static xSemaphoreHandle output_semaphore;

void output_task(void *params) {
    for (;;) {
        if (xSemaphoreTake(output_semaphore, portMAX_DELAY)) {
            P6OUT |= 0x80; /* Set pin 6.7 high. */
            vTaskDelay(PULSE_WIDTH);
            P6OUT &= ~0x80; /* Set pin 6.7 low. */
        }
    }
}

interrupt(PORT1_VECTOR) msp430_port1_vector(void) {
    P1IFG = 0x00;   /* Clear intrrupt flag for pin 1.7. */
    if (xSemaphoreGiveFromISR(output_semaphore, pdFALSE))
        taskYIELD();
}

int main(void) {
    xTaskHandle output;

    P1SEL   &= ~0x80; /* Configure pin 1.7 as digital I/O. */
    P1DIR   &= ~0x80; /* Configure pin 1.7 as input. */
    P1IE    |= 0x80;  /* Enable interrupts for pin 1.7. */
    P6DIR   = 0x80;   /* Configure pin 6.7 as output. */

    vSemaphoreCreateBinary(output_semaphore);
    xTaskCreate(output_task, "output", 128, NULL, tskIDLE_PRIORITY + 1, &output);
    vTaskStartScheduler();
}
```

6 Conclusions and Future Work

Timber allows the real-time behavior of embedded systems to be modeled by means of *reactive objects*. In this paper we have demonstrated how the Timber semantics translates to embedded real-time programming in C. This is realized through the implementation of TinyTimber (TT), a C API to a minimalistic Timber Run-Time system. The TT kernel has been introduced and performance metrics have been presented for a number of representative embedded platforms. Compared to a traditional thread based real-time OS (FreeRTOS), TT is shown to excel with its simple API and high timing accuracy. Future work includes improvements of responsiveness and timing accuracy by amortizing queue management and shortening the kernel critical section. Furthermore, the applicability of TT to severely memory constrained systems may be broadened by adopting SRP based scheduling. The generation of pre-emption levels directly from the specification (Timber/C using TT) is currently under investigation.

References

[1] Ambient Systems - for low cost, low power, wireless mesh networking solutions. `http://www.ambient-systems.net/`, 2006.

[2] FreeRTOS-A Free RTOS for ARM7,ARM9,Cortex-M3,MSP430,MicroBlaze,AVR,x86,PIC18,H8S,HCS12 and 8051. `http://www.freertos.org/`, 2006.

[3] TinyOS Community Forum —— An open-source OS for the networked sensor regime. `http://www.tinyos.net/`, 2006.

[4] T. P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[6] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.

[7] F. Boussinot, G. Doumenc, and J. Stefani. Reactive Objects. *Annals of Telecommunications*, 51(9-10):459–473, 1996.

[8] F. Boussinot and J.-F. Susini. The SugarCubes Tool Box: A Reactive Java Framework. *Software – Practice and Experience*, 28(14):1531–1550, Dec. 1998.

[9] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of Timber. In *APLAS 2003*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003.

[10] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *First IEEE Workshop on Embedded Networked Sensors*, 2004.

[11] J. Eriksson and P. Lindgren. A Comprehensive Approach to Design of Embedded Real-time Software for Controlling Mechanical Systems. In *The 14th Asia Pacific Automotive Engineering Conference , APAC14*, 2007.

[12] H. Klapuri, J. Takala, and J. Saarinen. Safety, liveness and real-time in embedded system design. *Journal of Network and Computer Applications*, 22(2):69–89, 1999.

[13] V. Leijon, P. Lindgren, and J. Eriksson. FIFO WiDOM: Timely Control Over Wireless Links. In *IEEE Multi-conference on Systems and Control, Singapore*, 2007.

[14] P. Lindgren, J. Nordlander, and J. Eriksson. Robust Real-Time Applications in Timber. In *Sixth IEEE International Conference on Electro,Information Tech, EIT*, 2006.

[15] J. Nordlander. *Reactive Objects and Functional Programming*. Phd thesis, Department of Computer Science, Chalmers University of Technology, Gothenburg, 1999.

[16] J. Nordlander, M. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, April 2002.

[17] J. Peterson. The Haskell Home Page. `http://haskell.org`, 1997.

Paper B

Uniform scheduling of internal and external events under SRP-EDF

Authors:
Simon Aittamaa, Johan Eriksson, and Per Lindgren

Contribution:
In this paper, my contribution includes discussion and analysis of possible solutions for handling external events, implementation and description of the example application, developing methods for obtaining performance measurements and performing the measurements, and writing the text of the paper.

Reformatted version of paper accepted for publication in:
Real-Time and Embedded Systems (RTES) 2010

Uniform scheduling of internal and external events under SRP-EDF

Simon Aittamaa, Johan Eriksson, Per Lindgren

Abstract

With the growing complexity of modern embedded real-time systems, scheduling and managing of resources has become a daunting task. While scheduling and resource management for internal events can be simplified by adopting a commonplace real-time operating system (RTOS), scheduling and resource management for external events are left in the hands of the programmer, not to mention managing resources across the boundaries of external and internal events. In this paper we propose a unified system view incorporating earliest deadline first (EDF) for scheduling and stack resource policy (SRP) for resource management. From an embedded real-time system view, EDF+SRP is attractive not only because stack usage can be minimized, but also because the cost of a pre-emption becomes almost as cheap as a regular function call, and the number of pre-emptions is kept to a minimum. SRP+EDF also lifts the burden of manual resource management from the programmer and incorporates it into the scheduler. Furthermore, we show the efficiency of the SRP+EDF scheme, the intuitiveness of the programming model (in terms of reactive programming), and the simplicity of the implementation.

1 Introduction

Embedded real-time systems are naturally defined as time-bound reactions to external and internal events. The correctness of hard real-time systems relies on executing all reactions in accordance to their time-bounds. In addition to meeting the reaction deadlines, system resources need to be adequately managed. Embedded software plays an increasingly important role for the realization of such systems. To aid system development, resource management and scheduling can be simplified by the use of a real-time operating system (RTOS). However, commonplace RTOSs treats external and internal events in a non-uniform manner, both with respect to scheduling and resource management[4, 10, 9]. The scheduling of internal events (managed by the RTOS) is generally overruled by the underlying hardware interrupt mechanism (scheduling reactions to external events). The obvious effect is that scheduling is non-uniform between internal and external events, complicating both system design and analysis. However, more importantly is that the resource management provided by the RTOS is in effect set out of play. All resources that might be accessed by external event reactions (interrupt handlers) must be explicitly protected. This forces the programmer to manually manage critical sections (by interrupt masking) whenever resources claimed might be accessed by external events. This requires

the programmer to diverge from the uniform system view and severely complicates the programming of real-time systems.

In this paper we present a method for scheduling and resource management that allows external and internal events to be treated uniformly from a programmers perspective. Our proposed solution deploys earliest deadline first (EDF) scheduling, and manages shared resources under the stack resource policy (SRP). Earliest Deadline First (EDF) scheduling has been shown to be optimal, given that there exists no shared resources [8]. In section 2 we introduce a collaborative hardware/software scheme that performs *pure* EDF (pure in the sense that both external and internal events are scheduled uniformly) scheduling onto platforms featuring static priority based interrupt hardware.

In addition to meeting the reaction deadlines, system resources need to be adequately managed. The stack resource policy is a priority ceiling protocol with the following properties [2]:

- schedulability test for systems with shared resources,
- deadlock free execution,
- resource management is incorporated into scheduling,
- task scheduling onto a single execution stack,
- pre-emption becomes as cheap as procedure calls,
- eliminates yielding and context switch overhead, and
- SRP gives a tight bound for priority inversion.

In conclusion SRP brings a set of sought after features for resource constrained real-time systems. The deadlock free execution increases system robustness, while the single stack execution eliminate the need (and memory overhead) of multiple execution stacks and multiple execution contexts. This in turn eliminates yielding and context switch overhead, since a task is never started unless all resources are available for the task to complete. Moreover, pre-emption becomes as cheap as a procedure call, since there are no context switches in a SRP scheduled system. Finally, the bounded priority inversion ensures system responsiveness.

2 EDF scheduling

In the following we assume that each task is triggered by an event. Each task has an absolute *baseline* (time of release), and an absolute *deadline*, the time in between gives a permissible execution window for the task. The intuition behind earliest deadline first scheduling is simple, tasks should be executed according to their deadline. This corresponds to scheduling the top element of a priority queue, holding all released tasks, sorted by their absolute deadline. Whenever a new event occurs, the corresponding task

is released, its absolute deadline should be computed, and the task should be inserted in the priority queue accordingly. Typically a task release can stem either from external or internal events. Internal events may be postponed to occur at a later point in time, facilitating e.g., periodic events. Common-place micro controllers support efficient hardware scheduling of external and timer events through interrupt handlers. However, the scheduling policies of interrupt handlers are pre-dominantly adopting static priority schemes, which place a major hurdle to efficient implementation of EDF scheduling.

We address this problem by introducing a collaborative hardware/software kernel scheme (pure EDF) that deploys EDF scheduling on both internal and external events.

2.1 Design criteria

The following key design criteria should be met:

- pure EDF scheduling of both external and internal events
- high timing accuracy (high timer granularity and bounded timer jitter)
- low overhead

2.2 Kernel anatomy for pure EDF

In the following we outline the mechanisms of a platform independent EDF kernel in accordance to the discussed design criteria.

The task structure has the following selectors:

.baseline - absolute point in time

.deadline - absolute point in time

.relativeDeadline

.code

Key components are:

- state variables
  - *rq* - ready queue, ordered by absolute deadline in ascending order
  - *tq* - timer queue, ordered by absolute baseline in ascending order
  - *dl* - the currently shortest absolute deadline
- kernel operations
  - *dispatch(t)*

    if $t.deadline < dl$ then {

predl = dl - push current deadline
dl = t.deadline
t.code() - execute task
dl = predl - pop pre-empted deadline
if *rq* != {} then *dispatch(rq.dequeue())*
} else *rq.enqueue(t)*

– *postpone(t)*
tq.enqueue(t)
timer.schedule(t.baseline)

– *interrupt(i)*
t = interruptTaskVector[i]
t.baseline = timer.getTime()
t.deadline = t.baseline + t.relativeDeadline
dispatch(t)

– *timer.interrupt*
while *tq.top().baseline <= timer.getTime()* {
r = tq.dequeue()
rq.enqueue(t)
} if *tq* != {} then
timer.schedule(tq.top().baseline)
dispatch(rq.dequeue())

- *rq.enqueue(t)*
- *tq.enqueue(t)*
- *rq.dequeue(t)*
- *tq.dequeue(t)*

In the following we will elaborate on the general considerations needed for meeting the stated criteria by a re-entrant kernel:

External events

For a pure EDF scheduler *all* scheduling decisions should be made based on basis of absolute deadlines. Hence, we need a mechanism to capture the absolute arrival times of external events, giving the baseline for the event and the corresponding task. (The baseline for internal events can be derived from the originating external event as discussed later.) In some cases we can rely on the underlying hardware to perform time-stamping of external events, however, in general hardware support is as best limited to a subset

of the event sources. A generic solution is to perform time-stamping in the interrupt handler (*interrupt(i)*). The accuracy is in this case highly dependent on the blocking time of the interrupt handler, hence all *interrupts* are handled in a pre-emptive fashion.

Internal events

Internal events, may emanate either directly from the execution of a task, or being postponed to be released at a future point in time (given a postponed baseline, relative to that of the originating task). Hence we need mechanisms to directly dispatch (*dispatch(t)*) and postpone events (*postpone(t)*). For realizing the latter case, we use the micro-controller hardware timer set to trigger an interrupt, that in turn will schedule the *timer.interrupt* task to release the postponed event.

Timer management

The timing accuracy is directly dependent on the operating frequency of the timer. In tick based kernels, timing events occur periodically, causing pre-emption overhead to the currently executing task - hence system load will increase with increased timing accuracy [4, 7]. Fortunately, most modern micro-controllers offer remedy by means of output-compare functionality. For such platforms we may use a free running timer, and set the absolute time for an interrupt to be scheduled (*timer.schedule(t)*).

Absolute time representation

As the timer has a finite representation (number of bits), the free running timer will eventually overflow (wrap around), such giving a truncated representation of the absolute point in time. Under the condition that the number of timer bits is sufficient to encode the largest baseline offset we may undertake this truncated view of time. If we need a larger time span for baseline offsets, a virtual timer can be deployed that extends the hardware timer (least significants bits) with arbitrary number of most significant bits managed in software, implemented e.g., by simply advancing the most significants bits on a timer overflow. In the case that the hardware timer bits suffice, the only effect of increasing the timing accuracy is that the range of baseline offsets is decreased (this, without the performance penalty of a fine granularity tick based system). In case we need to cope with larger baseline offsets, the overhead is limited to that of the virtual timer implementation and the additional cost of accounting for the range of the virtual timer on related operations.

It should be noted that his scheme does not guarantee any bound on the time-stamping error, this must be ensured by the interrupt source or in software, e.g. by disabling interrupt sources until they are allowed to trigger again.

Priority queue management

Most kernel primitives must perform some queue management, thus the performance of the kernel is directly related to the efficiency of the queue management. While this

can be done in several ways, we have chosen to use a simple linear queue for clarity. A heap-based data-structure would be more suitable for larger task-sets.

Example

Assume the following system configuration:

- *v=[timerTask, t1]*
- *rq*={}, *tq*={}
- $dl = \infty$
- *timerTask.relativeDeadline = 2*
- *timerTask.code = timer.interrupt*
- *t1.relativeDeadline = 7*
- *t1.code = ..., postpone(t2), ..., dispatch(t3)*
- *t2.baselineOffset = 4*
- *t2.relativeDeadline = 2*
- *t3.baselineOffset = Inherit*
- *t3.relativeDeadline = Inherit*

Initially the systems runs a non-terminating idle task with infinite deadline (hence *dl* is ∞). At this stage the system is possibly in low power mode awaiting stimuli. A task *t1* with relative deadline 7 is bound to interrupt source *s1*. Task execution, pre-emption, and permissible execution window is shown in figure 1. Stack allocation is shown in figure 2.

At time 2, an external event *s1* occurs, the corresponding interrupt handler is invoked by hardware, pre-empting the idle task. The baseline is set to 2, and the absolute deadline is set to 9. Task *t1* is dispatched and since it has the earliest deadline, *dl* is set to 9, stack resources are allocated, and *t1* is executed. Assume that *t1* first creates the postponed task *t2* with a relative deadline of 2, and a baseline offset of 4. This task is queued in the timer queue (*tq*) and the next timer interrupt is scheduled to occur at time 6. Then *t1* dispatches task *t3*, under the inherited base- and dead-line of *t1*, enqueuing it into the ready queue (*rq*).

At time 3, *t1* terminates, *dl* is restored to ∞, *t3* is dequeued from *rq* and dispatched. Since *dl* is ∞, *dl* is set to 9 and *t3* is executed.

At time 6, the timer interrupt pre-empts *t3*, sets baseline to 6, deadline to 8, and dispatches the *timerTask*. Since the *dl* is 9, *dl* is set to 8 and *timer.interrupt* is executed. Task *t2* is dequeued from *tq* and enqueued into *rq*. Since *tq* is now empty no further

timer interrupt is scheduled at this time. *timer.interrupt* returns to dispatch, the previous deadline (9) is restored, and the first task in *rq* is dispatched. Since *dl* is 9, *dl* is set to 8, stack resources are allocated and *t2* is executed.

At time 7, *t2* terminates, stack resources are deallocated and the previous deadline is restored (9). Since *rq* is empty the dispatch exists, and the pre-empted *t3* is resumed. At time 8, *t3* terminates, stack resources are deallocated and the previous deadline is restored (∞). Since *rq* is empty, dispatch exists and the interrupt handler for *s1* terminates, resuming the pre-empted idle task.

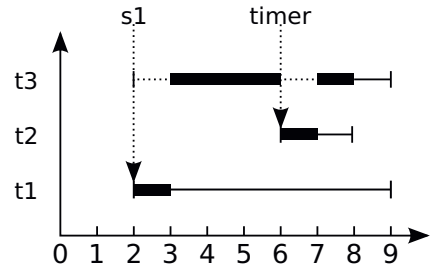

Figure 1: Example task execution, pre-emption, and permissable execution windows.

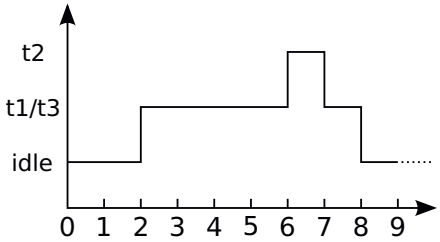

Figure 2: Example stack usage.

3 SRP scheduling under EDF

The EDF scheduler discussed in section 2 can easily be extended to support shared resources with the Stack Resource Protocol. There are only a few extensions that need to be made compared to the EDF Scheduler (section 2). Here we outline the changes to the previous implementation:

The task structure is extended with the following selector:

.preemptionLevel

The added/modified key components are:

- added state variables
  - *sc* - the system ceiling
- modified/added kernel operations
  - *dispatch(t)*

    if *t.deadline* $<$ *dl* AND *t.preemptionLevel* $<$ *sc* then {
    predl = *dl* - push current task
    dl = *t.deadline*
    sync(t) - execute task and claim resource
    dl = *predl* - pop pre-empted task
    if *rq* != {} then *dispatch(rq.dequeue())*
    } else *rq.enqueue(t)*
  - *sync(t)*

    saved_ceiling = *sc*
    sc = *t.preemptionLevel*
    t.code()
    sc = *saved_ceiling*

The fundamental idea with SRP is to allow resource-sharing in a well defined manner. Since we have resources we must in some way ensure mutual-exclusion, this is reflected by the addition of the *sync()* primitive. The *sync()* primitive ensures that the correct system ceiling is maintained. It should be noted that (as seen in the modified *dispatch(t)* pseudo-code) a lower pre-emption level means higher priority.

4 Implementation

4.1 Kernel

The kernel implementation shown in Listing 2 and 1 is a minimalistic implementation of SRP-EDF and a derivative of TinyTimber as described in [7]. One important note is that in section 3 resources are equivalent to tasks. However, in the kernel implementation, tasks and resources are separate data-structures.

4.2 Example Application

The example application given in Listing 3 is an implementation of the previously described example in section 2.2 for SRP-EDF.

Listing 1: kernel-source.h

```
// Implementation dependant MACROS
#define NTASKS          // Tasks (application dependant)
#define DISABLE()       // Disable interupts globally
#define ENABLE()        // Enable interupts globally
#define STATUS()        // Get interrupt status
#define TIMERGET(x)     // Get cur time
#define TIMERSET(x)     // Set next compare event
#define RETURN_TO(x)
 // Push a new function on the thread stack for the ISR
 // to return from. Used since we cannot call user code
 // from interrupt handlers on most platforms
#define SLEEP()         // Enter sleep mode
#define SEC(x)          // Seconds to time ticks
#define MSEC(x)         // Millisec to time ticks
#define USEC(x)         // Microsec to time ticks
#define initObject(x)   // Initialize object with
                        // pre-emption level

typedef struct {
  int pl;
} Object;
```

Listing 2: kernel-source.c

```
struct task_block {
  Task *next;       // for use in linked lists
  Time baseline;    // event time reference point
  Time deadline;    // absolute deadline (=priority)
  Object *to;       // receiving object
  int (*code)(int); // code to run
  int arg;          // argument to the above
};

struct task_block       tasks[NTASKS];
Task taskPool           = tasks;
Task taskQ              = NULL;
Task timerQ             = NULL;
Time timestamp          = 0;
Task cur_task           = NULL;
int system_ceiling      = MAX_INT;

static void enqueueByDeadline(Task p, Task *queue) {
    Task prev = NULL, q = *queue;
    while (q && (q->deadline - p->deadline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL)
        *queue = p;
    else
        prev->next = p;
}

static void enqueueByBaseline(Task p, Task *queue) {
    Task prev = NULL, q = *queue;
    while (q && (q->baseline - p->baseline <= 0)) {
        prev = q;
        q = q->next;
    }
    p->next = q;
    if (prev == NULL)
        *queue = p;
    else
        prev->next = p;
}

static Task dequeue(Task *queue) {
    Task m = *queue;
    *queue = m->next;
    return m;
}

static void insert(Task m, Task *queue) {
    m->next = *queue;
    *queue = m;
}
```

```
void dispatch(void) {
  DISABLE();
  if ( taskQ &&
     ( (cur_task == NULL) ||
      ( (cur_task->deadline < taskQ->deadline)
        && (system_ceiling > taskQ->to->pl)
      )
     )
  ) {
    Task saved_task = cur_task;
    cur_task = taskQ;
    taskQ=taskQ->next;
    // In the paper sync is inlined
    sync(cur_task->to, cur_task->code, cur_task->arg);

    insert( cur_task, &taskPool ); // recycle task
    cur_task = saved_task;
  }
  ENABLE();
}

void TIMER_INTERRUPT_HANDLER(void) {
  Time now;
  TIMERGET(now);

  while (timerQ && (timerQ->baseline - now < 0))
    enqueueByDeadline( dequeue(&timerQ), &taskQ );
  if (timerQ)
    TIMERSET(timerQ->baseline);

  RETURN_TO(taskswitcher);
}

Task intsched(Time dl, Object *to,
              int (*code)(int), int arg) {
  Task m;
  Time now;

  TIMERGET(now);

  m = dequeue(&taskPool);
  m->to = to;
  m->code = code;
  m->arg = arg;
  m->baseline = now;
  m->deadline = dl + m->baseline;
  m->rel_deadline = dl;

  enqueueByBaseline(m, &timerQ);

  TIMERSET(timerQ->baseline);
  RETURN_TO(taskswitcher);
  return m;
}

Task postpone(Time bl, Time dl, Object *to,
              int (*code)(int), int arg) {
  Task m;
  Time now;
  DISABLE();

  m = dequeue(&taskPool);
  m->to = to;
  m->code = code;
  m->arg = arg;

  // Negative values => INHERIT
  m->baseline = bl < 0 ? cur_task->baseline
                       : cur_task->baseline;
  m->deadline = dl < 0 ? cur_task->deadline
                       : m->baseline + dl;
  m->rel_deadline = dl; // Used for preemtion levels
  enqueueByBaseline(m, &timerQ);
  TIMERSET(timerQ->baseline);

  ENABLE();
  return m;
}

// Extention, used for syncronous requests
// to other objects, not discussed in paper
// but used for sharing resources.
int sync(Object *to, int (*code)(int), int arg) {
  int result;
  int saved_ceiling_stacked;
  int status = STATUS();

  DISABLE();
```

```
  saved_ceiling_stacked = system_ceiling;
  system_ceiling = to->pl;

  ENABLE();
  result = code(to, arg);
  DISABLE();

  system_ceiling = saved_ceiling_stacked;

  // Try to dispatch any preemted events
  taskswitcher();
  if (status) ENABLE();
  return result;
}

void initialize(void) {
  // Set up the timer, taskpool etc,
  // Implementation dependent
}

void idle(void) {
  ENABLE();
  while(1) SLEEP();
}
```

Listing 3: application-source.c

```
#include "LPC17xx.h"
#include "uTimber.h"

typedef struct{
    Object obj;
    // Internal state variables here.
} myobj;

myobj objt2 = { initObject( _OBJT2_PL_ ) };
myobj objt3 = { initObject( _OBJT3_PL_ ) };

myobj eintobj = { initObject( _EINT0_PL_ ) };

int t1(myobj *, int);
int t2(myobj *, int);
int t3(myobj *, int);

int t1(eintobj *self, int arg){
    POSTPONE(MSEC(4), MSEC(2), &obj2, t2);
    POSTPONE(-1, -1, &obj3, t3); // Inherit bl and dl.
}

int t2(myobj *self, int arg){
    // Perform 1 ms of work.
}

int t3(myobj *self, int arg){
    // Perform 4 ms of work.
}

void EINT0_IRQHandler(void){
// Level triggered interrupt handlers needs to be disabled
// in the hardware interrupt handler, otherwise we create
// a infinite loop. The task in responsible for enabling it
// again after servicing interrupt (if appropriate).
    NVIC_DisableIRQ(18);
// Schedule task, with a deadline of 7 ms.
    intsched(MSEC(7),&eintobj, t1, 0);
}

void main(void){
    initialize();
    NVIC_EnableIRQ(18);
    idle();
}
```

5 Experimental results

5.1 Platform, Compiler and Setup

The platform used when measuring was an NXP LPC1769 [1] featuring an Cortex-M3 MCU, 512k Flash, and 64k SRAM. The GNU Compiler Collection (GCC) version 4.4.3

was used to compile the test-code. All code was compiler with the compiler flags *-mcpu=cortex-m3 -mthumb -O2*, the *-mthumb* switch is required since the Cortex-M3 core only support the Thumb-2 instruction set. All code was run from the flash and the flash accelerator module was disabled.

5.2 Memory Requirements

The memory requirement of the kernel, is as follows:

- Code-size 644 bytes (Flash)
- Data-size 20 bytes (SRAM)
- Task-size $24 * n$ bytes (SRAM)
- Object-overhead $4 * m$ bytes (SRAM)

Where n is the total number of tasks and m is the total number of objects. All memory requirements are derived from the compiled code.

5.3 Timing Behaviour

The timing behaviour of the kernel was measured in clock-cycles by sampling the RIT timer of the LPC1769.

In the first series of test seen in table 1 all assume the best-case. That is, both the external and internal events have the earliest deadline, the resources are available, and the only a single event occurs. However, the synchronous call is always a constant-time operation. The number of cycles measured is defined as follows:

Internal Event The number of cycles between the release-time of the task and the execution of the first instruction of the task.

External Event The number of cycles between triggering of an interrupt and the execution of the first instruction of the interrupt-task.

Synchronous Call The number of cycles between invocation of the sync method and execution of the first instruction of the code argument.

To give an example of how the queue length impacts the timing behaviour we study a series of worst-case consecutive invocations of the *postpone(t)* primitive. The results from the test can be seen in table 2. The same worst-case behaviour can be expected for all kernel primitives that incorporate queue-handling. The accuracy of time-stamping can be made free of artifacts due to blocking (dominated by the queue-handling) if the interrupt hardware supports timer capture for external events (interrupts). In our simplistic prototype implementation we can clearly see the linear effect of using a linked list in the queue-handling. Other data structures, such as heaps, buckets, etc. provide a significant improvement for larger queue lengths, which leads to reduced queue-handling overhead, as well as improved accuracy of time-stamping in software (due to reduced blocking time).

Table 1: Runtime of kernel primitives.

Kernel Primitive	Clock Cycles
Internal Event	138
External Event	123
Synchronous Call	31

Table 2: Runtime of postpone(t) *primitive, with the given taskQ length.*

Queue Length	Clock Cycles
Empty	127
Length 1	136
Length 2	147
Length 3	160
Length 4	172

6 Related Work

Scheduling policies have been extensively studied from theoretical perspectives, and are at least for single core/single CPU systems to be considered as being well understood [11]. However, in practice results apply only if the model used for analysis corresponds to the system at hand. Work on scheduling theory often undertakes an ideal system model, neglecting scheduling overhead and interrupt handling. In [6] the cost of additional interrupt handling is included in the feasibility and schedulability test. However, in their model interrupt handlers are treated separately from application tasks (interrupts being scheduled at a higher priority). Our model differs in that interrupt handlers are indeed treated as being part of the application, where the occurrence of an interrupt corresponds to the release of a task. This integrated task and interrupt management model can also be found in [3]. However, in our work we focus on resource constrained systems under EDF and extend the results to EDF-SRP scheduling for systems with shared resources. In the context of stack based EDF schedulers, we also find AmbientRT [5]. However, under AmbientRT external events are treated separately from the application task set.

7 Conclusions and Future Work

With the outset that embedded real-time systems are naturally defined as time-bound reactions to external and internal events, EDF scheduling is a natural choice. We have developed a scheme that through efficient software scheduling of interrupts accomplish pure EDF even on commonplace platforms that deploy static priority scheduling of interrupt handlers. Furthermore, we have show how the pure EDF scheme can be extended to perform SRP under EDF. This gives us efficient shedulability test, deadlock free single stack execution, efficient pre-emption management and tightly bound priority inversion. We have shown the efficiency of the proposed schema quantified by experiments on our prototype implementations.

Future work include investigating the possibility of hardware support for EDF+SRP scheduling of internal and external events, as well as time-stamping of external events (interrupts). We are also working on a complete system analysis (incorporating interrupt overhead, queue-handling overhead, blocking time, worst-case execution time, etc.) for SRP+EDF scheduled system. As a part of the complete system analysis we are investigating the impact of the underlaying data-structures relating to the queue-handling overhead and blocking time.

References

[1] 32Bit ARM Cortex-M3, NXP LP1769. http://www.nxp.com/pip/LPC1769_68_67_66_65_64_4.html.

[2] T. P. Baker. A stack-based resource allocation policy for realtime processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[3] L. E. L. del Foyo, P. Mejia-Alvarez, and D. de Niz. Predictable interrupt scheduling with low overhead for real-time kernels. *Real-Time Computing Systems and Applications, International Workshop on*, 0:385–394, 2006.

[4] FreeRTOS-A Free RTOS for ARM7, ARM9, Cortex-M3, MSP430, MicroBlaze. http://www.freeRTOS.org.

[5] T. Hofmeijer, S. Dulman, P. Jansen, and P. Havinga. Ambientrt - real time system software support for data centric sensor networks. In *Proceedings of the 2004 Intelligent Sensors, Sensor Networks and Information Processing Conference*, pages 61–66. IEEE Computer Society Press, 2004.

[6] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. pages 212–221, 1993.

[7] P. Lindgren, J. Eriksson, S. Aittamaa, and J. Nordlander. Tinytimber, reactive objects in c for real-time embedded systems. In *DATE*, pages 1382–1385. IEEE, 2008.

[8] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[9] O. V. Portal. http://www.osek-vdx.org.

[10] D. C. Sastry and M. Demirci. The qnx operating system. *Computer*, 28(11):75–77, 1995.

[11] J. A. Stankovic, M. Spuri, M. D. Natale, S. S. S. Anna, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28:16–25, 1995.

Paper C

Scheduling Garbage Collection in Real-time Systems

Authors:
Martin Kero and Simon Aittamaa

Contribution:
In this paper, my contribution includes modifying the Timber run-time system to enable performance measurements, modifying the garbage collector, discussion of ideas and analysis of the experimental results, as well as describing the testing setup.

Reformatted version of paper accepted for publication in:
The International Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS) 2010

Scheduling Garbage Collection in Real-time Systems

Martin Kero, Simon Aittamaa

Abstract

The key to successful deployment of garbage collection in real-time systems is to enable provably safe schedulability tests of the real-time tasks. At the same time one must be able to determine the total heap usage of the system. Schedulability tests typically require a uniformed model of timing assumptions (inter-arrival times, deadlines, etc.). Incorporating the cost of garbage collection in such tests typically requires both artificial timing assumptions of the garbage collector and restricted capabilities of the task scheduler. In this paper, we pursue a different approach. We show how the reactive object model of the programming language Timber enables us to decouple the cost of a concurrently running copying garbage collector from the schedulability of the real-time tasks. I.e., we enable *any* regular schedulability analysis without the need of incorporating the cost of an interfering garbage collector. We present the *garbage collection demand analysis*, which determines if the garbage collector can be feasibly scheduled in the system.

1 Introduction

As capabilities of embedded devices increase, the possibilities and demands of more complex systems arise. This leads to an ever increasing need of more sophisticated run-time system features. The ability to use shared dynamic data and parallel/concurrent execution of tasks are both examples of features that enables the programmer to more extensively utilize these new capabilities. However, the two features in conjunction makes automatic memory management (garbage collection) a necessity.

Garbage collection has widely been acknowledge for reducing development time as well as enhancing reliability of software systems. In the real-time system context, the work on garbage collection can roughly be divided into two categories. The first category concerns the development of garbage collection algorithms suitable for real-time systems. This line of work was initiated already in the late 70's by Baker [2], and has since then received most of the attention. The second category, to which this paper belongs, concerns schedulability analyses of garbage collected real-time systems. This line of research was initiated by Henriksson [14].

A real-time system is defined in terms of a set of *tasks* for which timing assumptions are uniformly defined (inter-arrival times and patterns, deadlines, etc.). Corresponding schedulability tests crucially depend on the uniformed task model [21]. Adapting such tests to incorporate the cost of garbage collection typically imposes unnecessary restrictions on both task model and scheduler. The main issue is that schedulability analysis

requires execution time estimates of the tasks to be independent, whereas such estimate of a garbage collector depends on a combination of memory and timing behaviors of the real-time tasks.

In this paper, we pursue a different approach. We show how an incremental copying garbage collector [18] deployed as a concurrent process in the reactive environment of the programming language Timber [23] enables us to decouple the cost of garbage collection from the feasibility of the real-time tasks. I.e., we enable *any* schedulability analysis for a real-time system, without the need of incorporating the cost of garbage collection into that analysis. Instead, we offer another analysis, decoupled from the regular schedulability analysis. Given a schedulable set of tasks, our analysis provides a sufficient test to determine if the garbage collector can be scheduled in the system. We call this analysis *garbage collection demand analysis*. Our approach relies on the following two key properties:

1. Each atomic increment of the garbage collector can be executed within a small and constant time (including root-set scanning and synchronization operations).

2. In order to preserve schedulability of the task set, the garbage collector runs as the lowest priority process.

The running time of a concurrent copying garbage collector depends on the amount of live heap memory and the synchronization work due to interrupting mutators. Besides timing assumptions for the task set, our analysis requires upper-bounds on global live heap space and heap allocation of each task as input parameters. Such analyses does not fit within the scope of this paper. However, a forthcoming paper presenting live heap space analysis for real-time systems is in preparation [19].

The general motivation behind this paper is pervaded by the ambition to achieve provable schedulability properties of a real-time garbage collector, as opposed to improve performance. Although improving schedulability is important, it is not within the scope of this paper to embark that field. Instead, our objective is to enable schedulability tests for garbage collection independently of which scheduling policy that is employed for the real-time tasks.

2 Timber – reactive objects

Timber is a, strongly typed, object-oriented programming language for reactive real-time systems. The basis of the language is the *reactive object*. This section is a brief description of the parts of the language that are relevant to the rest of this paper. More in-depth descriptions can be found at the Timber official homepage [23], in the draft language report [5], the formal semantics definition [7], and the descriptions of the reactive object model [25, 24].

The reactive objects in Timber are concurrent. That means, activity in one object can execute concurrently with activity in other objects. Furthermore, objects are the only state carrying data structures (mutable data) and the only way to access the state

of an object is through its methods. State integrity is preserved by restricting methods of one particular object to execute under mutual exclusion. In order to preserve aliveness, methods cannot block-and-wait for future events. Events are instead interpreted as method calls and each method will eventually terminate, leaving the object in an idle state. Methods can either be asynchronous (equivalent to events in their own right) or synchronous rendezvous operations.

Timber offers a capability to express timing constraints directly in the model. Asynchronous methods (events) are associated with both a *baseline* (earliest release time) and a *deadline* (latest response time). If not explicitly set, the *baseline* of a method invocation is inherited from the method who called it. If the event is due to an interrupt, the baseline is set to the time-stamp of the interrupt. Deadlines are expressed relative to the current baseline. Similar to baselines, deadlines are inherited if not explicitly set.

Correct timing behavior of a method is that it must start executing and finish within its permissible execution window. Baselines and deadlines can be adjusted by explicitly expressing it in the definition of an asynchronous method (or by the caller).

Aside from the reactive objects, Timber consists of a expression layer based on a *pure* (no side-effects, only immutable data) call-by-value functional language.

3 The Timber run-time model

In order to meet the semantic specification of the language, a Timber program will depend on the infrastructure provided by the run-time kernel of the language. Conceptually, the kernel has to offer the following services:

- Create an object,
- Send a synchronous message, and
- Send an asynchronous message (possibly delayed).

In reality, creating an object maps to allocation of storage. The only difference between allocating an object and an immutable data structure (struct, tuple, cons cell, etc.) is that in order to enable mutual exclusion between methods of the same object, each object needs a *lock* field. A synchronous rendezvous is simply accomplished by locking the object before running the code of the method, and release the lock afterwards. An asynchronous method call on the other hand requires a new *message* (conceptually an execution thread) to be allocated and enqueued in the proper message queue (timer queue if delayed). Once this message gets to run, it simply makes a regular synchronous call.

The default scheduling mechanism in the kernel is a preemptive priority scheduler based on deadlines. Delayed messages are scheduled based on baselines. The scheduler interacts with three types of data; objects (includes locks), messages (including executable code, possibly parameters, a baseline, and a deadline), and threads (messages extended with an execution context). Objects are the shared resources for which messages acquire and release locks. Asynchronous messages that are latent (either delayed or pending)

are, when scheduled to run, promoted to a unique thread. The concurrency of a Timber program is thus accomplished by the scheduler; which, for each independent schedulable message, may allocate a new unique thread of execution.

The interface of the scheduler consists of four entry points.

1. Whenever a new asynchronous message is posted (either internally by another method or externally by an interrupt).

2. When a method calls lock or unlock for an object.

3. When a method terminates.

4. When a timer interrupt occurs.

The scheduler manages three main data structures. A stack of active threads (including, of course, each thread's individual stored execution context), a priority queue of pending messages, and a queue of delayed messages ordered by earliest baseline first.

There are a few very important properties of the kernel that are worth mentioning. All lock operations eventually return, either with the acquired lock or with a deadlock error. The lock of an object may only be owned by one message at a time. The highest priority thread, or the thread holding a lock wanted by the highest priority thread, will always be the one scheduled to run. All messages will run after their baselines since a delayed message will be promoted to the queue of pending messages once its baselines has expired. The aliveness property of a Timber program is crucially dependent upon the fact that a successful lock is always followed by an unlock and that all locks are acquired in a nested manner.

4 The GC algorithm

We will base our analysis on the incremental copying garbage collector presented in [18]. In this section we give a shortened description of the essential parts of the algorithm. For a complete specification with correctness proofs see the original presentation.

We use Cheney's in-place breadth-first traversal of gray objects [9], and we deploy a read barrier similar to that of Brooks [6]; i.e., reads to old copies in white space are forwarded to their corresponding new copies in the gray/black heap. Furthermore, we use a write barrier in the style of Steele [27], where the tri-color invariant is upheld by reverting black objects to gray upon mutation.

Let x, y, z range over heap addresses, and let n range over integers. Let u, v range over values and be either a heap address or an integer. Let U and V range over sequences of such values.

A heap node can be either a sequence of values (enclosed by angle brackets $\langle V \rangle$) or a single forwarding address (denoted $\bullet x$). A heap H is a finite mapping from addresses to

nodes, as captured by the following grammar.

$$\begin{array}{lrcl} \text{(heap)} & H & ::= & \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\} \\ \text{(node)} & o & ::= & \langle V \rangle \mid \bullet\, x \\ \text{(value)} & v & ::= & x \mid n \end{array}$$

The domain $dom(H)$ of a heap $H = \{x_1 \mapsto o_1, \ldots, x_n \mapsto o_n\}$ is the set $\{x_1, \ldots, x_n\}$. A heap look-up is defined as $H(x) = o$ if $x \mapsto o \in H$. We write U, V to denote the concatenation of the value sequences U and V. Along the same line, we write H, G for the concatenation of heaps H and G, provided their domains are disjoint.

The heap is described as a triple of subheaps separated by heap borders ($\mid$). A heap border has the same meaning as the regular concatenation operator for heaps, but it also provides necessary bookkeeping information. The three subheaps capture the white, gray, and black part of the heap as in the *tricolor abstraction* [10].

The algorithm is based on a labeled transition system (LTS) [22], where garbage collection transitions are so called internal (τ) transitions. Each individual τ transition constitutes an atomic increment by the garbage collector. In Figure 1 and 2, all possible internal transitions are shown. Determinism between different internal transitions are achieved by pattern matching, as each configuration matches only one single clause. The clauses are furthermore divided into two groups, which we call *scan* and *copy* transitions. A garbage collection cycle is a sequence of such transitions beginning with a Start transition and ending with a Done transition.

$$H_0 \overset{\text{Start}}{\longrightarrow} H_1 \longrightarrow \ldots \longrightarrow H_{n-1} \overset{\text{Done}}{\longrightarrow} H_n$$

Notice that an active garbage collection cycle is identified by a non-empty white subheap.

In contrast, the external transitions, such as mutations and allocations, are labeled, denoted by $H \xrightarrow{l} H'$. The definition of these transitions are shown in Figure 3. We use a single root pointer r to capture the root-set. Even though a real system most likely will contain more than one root, this can easily be captured in our model by adjusting the content of the node pointed to by r. I.e., the actual root-set is the content of the node labelled r.

At the beginning of a cycle the heap has the form $\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$, and initiating a garbage collection cycle (Start) invalidates the whole heap except for the root node. That is, all nodes but r are made white by placing them to the left of the white-gray heap border. The algorithm then proceeds by scanning gray nodes (ScanStart) and takes proper actions when embedded addresses are encountered. This is accomplished by a *scan pointer* that traverses the gray nodes. The scan pointer is denoted by the symbol $\downarrow$ and has similar function and purpose as the heap borders, i.e. regular concatenation as well as bookkeeping information. When a whole node has been scanned it is promoted from gray to black (ScanDone). When there are no more gray nodes to scan the garbage collector is finished (Done).

During scanning of a gray node, encountering an address may result in one of three possible actions. If the address found is not in the white heap, the algorithm just goes on to examine the next gray node field (SCANADDR). If the address is in the white heap, and the corresponding node is a forwarding node, the forwarding address replaces the encountered address (FORWARD). If, on the other hand, the node found is a regular white node, copying is initiated (COPYSTART). This is done by allocating a new empty node in the gray heap and *locking* the scan pointer, which we denote by the alternative concatenation symbol symbol $\uparrow_z$ (where the index is the address of the new empty node). The white node is then copied word by word (COPYWORD) until the whole node has been copied (COPYDONE). At this point, the address of the newly allocated node replaces the old encountered address, and the original white node is converted into a forwarding node.

START	$\emptyset \mid G, r \mapsto \langle V \rangle \mid \emptyset$	$\longrightarrow$	$G \mid r \mapsto \langle V \rangle \mid \emptyset$	
SCANSTART	$W \mid G, x^{\dagger} \mapsto \langle V \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V \rangle \mid B$	$W \neq \emptyset$, †may be dirty
SCANINT	$W \mid G, x \mapsto \langle V \downarrow n, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, n \downarrow V' \rangle \mid B$	$W \neq \emptyset$
SCANADDR	$W \mid G, x \mapsto \langle V \downarrow y, V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle V, y \downarrow V' \rangle \mid B$	$W \neq \emptyset$, $y \notin dom(W)$
SCANRESTART	$W \mid G, \dot{x} \mapsto \langle V \downarrow V' \rangle \mid B$	$\longrightarrow$	$W \mid G, x \mapsto \langle \downarrow V, V' \rangle \mid B$	$W \neq \emptyset$
SCANDONE	$W \mid G, x \mapsto \langle V \downarrow \rangle \mid B$	$\longrightarrow$	$W \mid G \mid x \mapsto \langle V \rangle, B$	$W \neq \emptyset$
DONE	$W \mid \emptyset \mid B$	$\longrightarrow$	$\emptyset \mid B \mid \emptyset$	

Figure 1: Scan transitions.

5 Scheduling the GC

The problem of scheduling garbage collection in real-time systems is twofold. The requirements put on the garbage collector scheduler are:

1. *the garbage collector must not cause any task to miss its deadline, and*
2. *the system must not run out of memory.*

One can easily see that fulfilling one of the requirements may cause a failure to meet the other. E.g., to avoid running out of memory, the garbage collector needs to run, which in turn may cause a task to fail meeting its deadline. This scheduling problem is not easily solved and it becomes even more difficult in our case because we use a copying collector, which, conceptually, has to finish before any garbage memory can be reused.

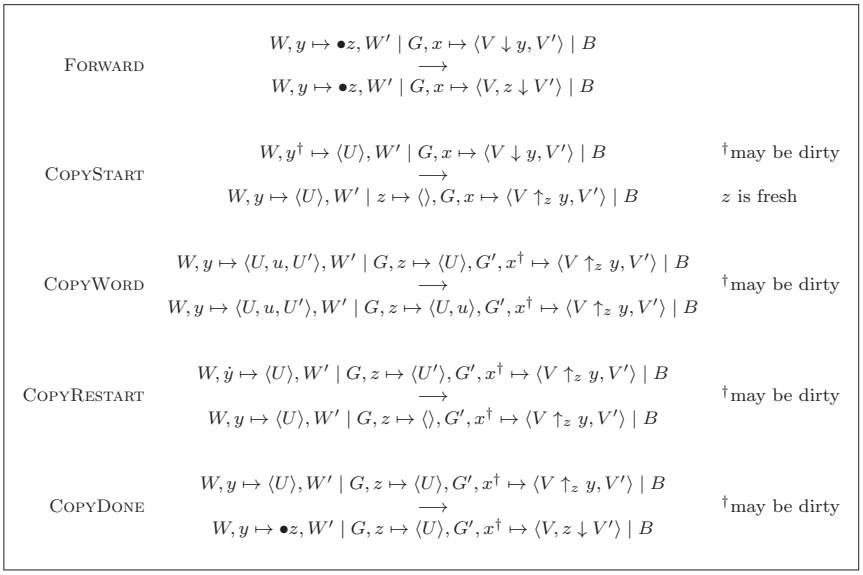

Figure 2: Copy transitions.

5.1 Idle time GC

In the general case, finding and scanning the root-set incrementally is a very difficult task. The problem is that the root-set is not constant. Finding all roots requires scanning of, not only static fields and CPU registers, but also the run-time stacks. Since the depth and content of the run-time stacks are not constant, the cost of scanning them is bound to be unpredictable. Scanning them incrementally is also deemed to be notoriously hard due to their volatile nature. In the worst case, the scanning process would need to be restarted at each increment since the content of the stacks may have been altered altogether. The idea of possibly restart the scanning process of an arbitrary sized root-set at each increment is not very appealing, especially not when it comes to real-time systems.

In order to remedy this problem, we will restrict the garbage collector to only run at idle time. Although this policy has been considered inferior to other approaches (in terms of performance), it comes with a couple of substantial advantages.

Constant time root-set scanning

In a reactive system, such as Timber, the root-set during idle time is very small. The queue of pending messages is empty and no run-time stacks exist. In fact, the root-set consists of only two elements, the interrupt vector and the queue of delayed messages. The size of these data structures are directly connected to the number of tasks defined. Since

MUTW	$H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \xrightarrow{w(p:i=q)} H' = W, \dot{x} \mapsto \langle U[i] := y \rangle, W' \mid G \mid B$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
MUTG	$H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \xrightarrow{w(p:i=q)} H' = W \mid G, \dot{x} \mapsto \langle U[i] := y \rangle, G' \mid B$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
MUTB	$H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \xrightarrow{w(p:i=q)} H' = W \mid \dot{x} \mapsto \langle U[i] = y \rangle, G \mid B, B'$	if $read(H, r, p) = x$ and $read(H, r, q) = y$
WRITE	$H, x \mapsto \langle U \rangle, H' \xrightarrow{w(p:i=n)} H, x \mapsto \langle U[i] := n \rangle, H'$	if $read(H, r, p) = x$
READ	$H \xrightarrow{r(p=n)} H$	if $read(H, r, p) = n$
ALLOCMUTW	$H = W, x \mapsto \langle U \rangle, W' \mid G \mid B \xrightarrow{a(p:i)} H' = W, \dot{x} \mapsto \langle U[i] := z \rangle, W' \mid z \mapsto \langle\rangle, G \mid B$	if $read(H, r, p) = x$
ALLOCMUTG	$H = W \mid G, x \mapsto \langle U \rangle, G' \mid B \xrightarrow{a(p:i)} H' = W \mid z \mapsto \langle\rangle, G, \dot{x} \mapsto \langle U[i] := z \rangle, G' \mid B$	if $read(H, r, p) = x$
ALLOCMUTB	$H = W \mid G \mid B, x \mapsto \langle U \rangle, B' \xrightarrow{a(p:i)} H' = W \mid \dot{x} \mapsto \langle U[i] := z \rangle, z \mapsto \langle\rangle, G \mid B, B'$	if $read(H, r, p) = x$

Figure 3: Mutator transitions.

periodic task releases are achieved by posting a delayed message for the next instance of the task, each periodic task will contribute with one message in the queue of delayed messages. Tasks that are connected to external events (hardware interrupts) has their corresponding messages in the interrupt vector. Thus, if the set of tasks is statically known, the size of the root-set is also statically known.

Schedulability analysis is preserved

Since the garbage collector is the lowest priority process in the system it will never compete for CPU time with the real-time tasks. Nonetheless, if the garbage collector is running (which must be assumed in the worst case) some extra overhead is put on the tasks due to synchronization issues. Furthermore, even though the garbage collector is incremental, the longest atomic increment must be taken into account as an execution

time overhead for each task.

In the next section we will look into the details of these overheads and show that the synchronization overhead, is very small (zero for the hard real-time case).

6 GC overhead

Restricting the garbage collector to only run at idle time reduces the problem of determining the time overhead due to garbage collection significantly. The induced overhead can be divided into two parts, the cost of actually interrupting the garbage collector while it is running, and the cost of synchronizing with garbage collector (read and write barriers).

6.1 Longest atomic increment of the GC

The longest atomic increment of the garbage collector is easily determined directly from the algorithm definition (Fig. 1 and 2). Since none of the transitions are dependent on any variable-sized data the cost is constant in terms of number of instructions to execute.

6.2 Synchronization

Generally, the need of synchronization for an incremental garbage collector is to preserve the tri-color invariant [10]. However, for a copying garbage collector the situation is a little bit trickier. Not only is the synchronization mechanism needed to preserve the liveness view of the heap but also for assuring that mutators access the new copies (if they exist) of heap objects. The synchronization between mutators and the garbage collector typically boils down to so called *barrier* methods. One for reading and one for writing heap data.

The reactive objects of Timber enable us to reduce these synchronizations substantially. The reason is that the only mutable data structures are the objects. Immutable data, as pointed out already by Doligez and Leroy [11] as well as Huelsbergen and Larus [15], require no synchronization actions.

Since access to objects must be made under mutual exclusion, a combined read/write barrier can be used that is invoked as part of the lock operation. That is, instead of calling a barrier method for every heap access, we now only need to invoke the barrier when an object is locked. Since most methods only lock one object, the combined read/write barrier is called only once per method invocation.

6.3 Hard real-time systems

Since an object should contain at least one method in order to be meaningful, the concept of dynamic object creation ultimately leads to dynamic creation of tasks. However, generally speaking, in order to determine schedulability, dynamic creation of tasks cannot be allowed. All tasks must be statically known. Thus creating objects dynamically

cannot be allowed. These mutable objects can then be allocated static. What is left in the dynamic heap is now only immutable data which do not require any synchronization. Synchronization overhead is hence eliminated altogether. In Fig. 4, an example of how such a system is arranged in memory is shown.

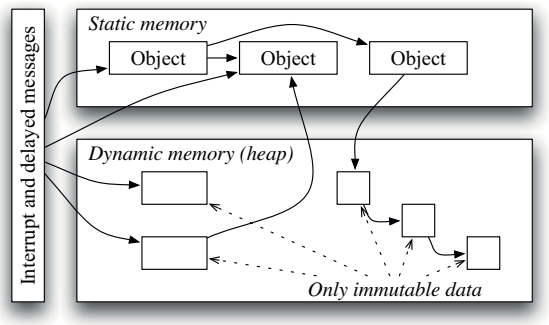

Figure 4: Example of a Timber system with only static objects.

7 Schedulability of the GC

We have so far shown that the first requirement of garbage collection scheduling is fulfilled by our collector. We have accomplished this by restricting the garbage collector to only run at idle time. In this section we will look at what is required in order to fulfill the second requirement.

7.1 Garbage collection demand analysis

Baruah et al. presents a feasibility test for real-time tasks called *processor demand analysis* [4, 3]. We present a feasibility test for garbage collection in real-time systems called *garbage collection demand analysis*. In contrast to checking that the processor demand for all possible time windows are less than the size of the window, our analysis determines the required size of the window such that the garbage collection demand is less than the size of the window. If such a window size can be found and the required memory demands of that window size can be met, our garbage collector can feasibly be schedule in the system.

We will only look at the hard real-time case for the analysis, which means no mutable data will be present on the heap and the transitions SCANRESTART and COPYRESTART will never be taken.

We begin by looking at the execution time of the garbage collector if it is allowed to run without interruptions, and then we look at the cost contributed by tasks interrupting

the collector.

The garbage collector transitions are defined in Fig. 1 and 2. The Start and Done transition is only taken once per garbage collection cycle. The number of transitions of each kind can be derived from three basic parameters of the heap to be collected:

$$\begin{aligned} M: &\ \text{Amount of reachable memory to copy (in words).} \\ O: &\ \text{Number of reachable nodes on the heap.} \\ R: &\ \text{Number of reachable references on the heap.} \end{aligned}$$

We can now formulate the execution time of garbage collection (T_{gc}) in terms of these parameters and execution times of each transition (denoted by $T_{<\text{name of transition}>}$).

$$\begin{aligned} T_{gc} =& M * T_{\text{CopyWord}} + \\ & O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} + T_{\text{CopyStart}} + \\ & T_{\text{CopyDone}}) + R * T_{\text{ScanAddr}} \\ & + (R - O) * T_{\text{Forward}} + T_{\text{Start}} + T_{\text{Done}} \\ =& M * T_{\text{CopyWord}} + \\ & O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} + T_{\text{CopyStart}} + \\ & T_{\text{CopyDone}} - T_{\text{Forward}}) + \\ & R * (T_{\text{ScanAddr}} + T_{\text{Forward}}) \\ & + T_{\text{Start}} + T_{\text{Done}} \end{aligned} \tag{1}$$

Due to the fact that the tasks never can cause anymore copying work for the garbage collector[1] and we only have immutable data on the heap, the only extra garbage collection cost of interruptions is due to new allocations. Similarly to the heap parameters above we have three parameters due to new allocations during garbage collection:

$$\begin{aligned} A_i^M: &\ \text{Amount of new memory allocated by task } i \text{ (in words).} \\ A_i^O: &\ \text{Number of nodes allocated by task } i. \\ A_i^R: &\ \text{Number of references allocated by task } i. \end{aligned}$$

We can now formulate the garbage collector execution time due to new allocations made by task i (T_i^A).

[1] This is because new data is allocated in tospace and garbage nodes can never become reachable again (see Lemma 5.1 in [18]).

$$\begin{aligned} T_i^A =& A_i^O * (T_{\text{ScanStart}} + T_{\text{ScanDone}}) + \\ & A_i^R * T_{\text{ScanAddr}} + \\ & (A_i^R - A_i^O) * T_{\text{Forward}} \\ =& A_i^O * (T_{\text{ScanStart}} + T_{\text{ScanDone}} - T_{\text{Forward}}) + \\ & A_i^R * (T_{\text{ScanAddr}} + T_{\text{Forward}}) \end{aligned} \tag{2}$$

Now we will look at the total garbage collection demand for a time window of size t. For the sake of simplicity we will assume that the heap and allocation parameters are constants. Even though this is not true in reality (reachable memory, allocation rates, etc. tend to vary over time) we can always assume the worst case. We will return to the validity of this assumption at the end of this section.

Let n be the number of tasks, T_i^A be the garbage collection execution time due to allocations made by task i, $\left\lceil \frac{t}{P_i} \right\rceil$ the maximum number of releases of task i in a time window of size t, C_i the execution time of task i, and P_i the period of task i. We can now formulate the total garbage collection demand (C_{gc}) in a time window of size t as follows.

$$C_{gc}(t) = T_{gc} + \sum_{i=1}^{n} \left\lceil \frac{t}{P_i} \right\rceil * (C_i + T_i^A) \tag{3}$$

Informally, C_{gc} consists of two independent parts. One accounting for the execution time required to garbage collect the heap, as if undisturbed. The second part accounts for the extra time induced and consumed by tasks interrupting the garbage collector.

Theorem 7.1. *For any t such that $C_{gc}(t) \leq t$ the garbage collector can be scheduled feasibly (w.r.t. time) with a period of t.*

Proof By contradiction. ∎

The memory needs of the system is formulated as a function of the period t of the garbage collector.

$$M_{tot}(t) = 2 * \left(M + \sum_{i=1}^{n} \left\lceil \frac{t}{P_i} \right\rceil * A_i^M \right) \tag{4}$$

In order to complete the proof of correctness of the feasibility test, we need to show that it is sufficient to test feasibility of the worst case of heap and allocation parameters.

Lemma 7.2. *If the garbage collector can feasibly be scheduled with a period of t for a system with heap parameters M, O, and R, and for all tasks i, allocation parameters A_i^M, A_i^O, and A_i^R. Then, for any $M' \leq M$, $O' \leq O$, $R' \leq R$, and for all tasks i, $A_i^{M'} \leq A_i^M$, $A_i^{O'} \leq A_i^O$, and $A_i^{R'} \leq A_i^R$, the garbage collector can still be feasibly scheduled with a period of t.*

Proof Follows directly from the fact that Equation 1 and 2 are monotonic. ∎

8 Experimental results

The objective of the experimental study is to confirm that the model conceptually captures the execution time of the garbage collector. In other words, for a particular platform, there shall exist constants (e.g. T_{CopyWord}) such that we can construct the platform specific model of worst-case garbage collection execution time by simply replacing the constants in the general model by appropriate values. Naturally, experimental results are by no means guaranteed to reveal the worst-case behavior (i.e. the constants associated with the actual worst-case behavior). However, if set up appropriately, it will inevitably expose flaws of the model (non-existence of constants). It should be noted that this experimental study is by no means a performance evaluation of the garbage collector or the way it is scheduled.

In the model, we have five parameters (M, O, R, A_i^O, and A_i^R) that affects the execution time of the garbage collector (T_{gc} and T_i^A). The general approach we pursue is to measure the effect of each parameter in isolation whilst keeping the other parameters constant.

8.1 Hardware platform

The experimental platform we use is the LPC2468 Developer's Kit from Embedded Artists [29], with the standard LPC2468-16 OEM board replaced by an LPC2468-32 OEM board. The LPC2468-32 OEM board contains a standard NXP ARM7TDMI-S LPC2468 microcontroller [30] along with 32 MB of SDRAM from Micron [28]. The reason we choose this platform is simply because it has a sufficient amount of memory (i.e. enables sufficiently wide ranges of input parameter values for the measurements), a JTAG interface, and a fairly simple memory hierarchy. The microcontroller has no cache, but there exists a primitive buffering mechanism within the External Memory Controller (EMC).

8.2 Software platform

In order to make it easy to run the measurements, we use two different code-bases. The first code-base is provided by Embedded Artists [29] and takes care of the initialization of the LPC2468, setting the CPU clock to 57.6 MHz and initializes the External Memory Controller (EMC). When the initialization of the LPC2468 is complete the microcontroller will be stuck in an infinite-loop. Once the infinite-loop is reached we connect to the microcontroller using the JTAG interface and upload the second code-base into the SDRAM. The second code-base consists of the test-code along with the Timber Run-Time System (Timber RTS). The Timber RTS used in the experiments is based on the standard ARM RTS available in the Timber darcs repository [23]. The ARM RTS has then been modified to support logging of several events within the RTS, among other

things the GC-time and execution-time of messages. Special care is taken to minimize the effects of the logging. While it is impossible to completely remove the effects of logging the overhead is very small, constant, and predictable. After the test-run is complete we download the log from the internal SRAM of the microcontroller via the serial port.

8.3 Measurements

Parameters affecting T_{gc}

We have three heap parameters affecting the resulting T_{gc}. We setup the test runs in such way that each parameter can be varied in isolation (except when varying number of live nodes). The general configuration is a time-triggered scheduling of the garbage collector. The live data on the heap is a constant structure which is relocated 100 times by the garbage collector for each parameter value. We collect the total running time of each garbage collection cycle. During these measurements, we do not have any tasks running concurrently with garbage collector. We do this procedure for 1000 different values of the parameter in question.

Varying amount of live memory We use one live node, which we vary the size from 1 to 1000 words containing no references. The results of this is shown in Figure 5.

Varying amount of live references We use one live node with a constant size of 1000 words, for which we vary the number of self references from 1 to 1000. The results of this is shown in Figure 6.

Varying number of live nodes We use a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this is shown in Figure 7. Due to the fact that we cannot easily generate equivalent data structures with amount of live memory and number live references constant and vary the number of nodes with sought granularity, we actually vary all three parameters in a controlled way (i.e. 16 words live memory, 1 live reference, and 1 live object per data point).

Parameters affecting T_i^A

We have two heap parameters affecting the resulting T_i^A. We setup the test runs in such way that each parameter can be varied in isolation. The general configuration is, again, a time-triggered scheduling of the garbage collector, but now accompanied with a task interrupting the GC once at every run. The amount of live data on the heap is kept constant over all runs as a constant payload of work to be done. We collect the total running time of each garbage collection cycle. We do this procedure for 1000 different values of the parameter in question.

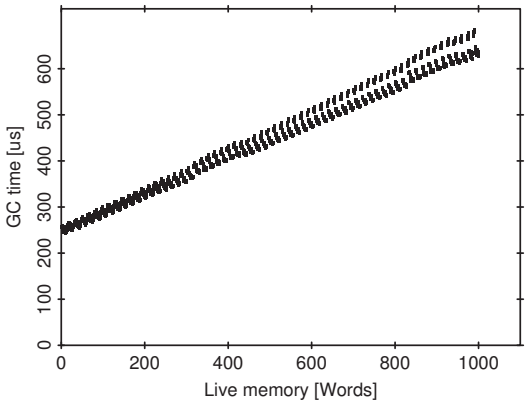

Figure 5: Measurement data of GC time as a function of live memory.

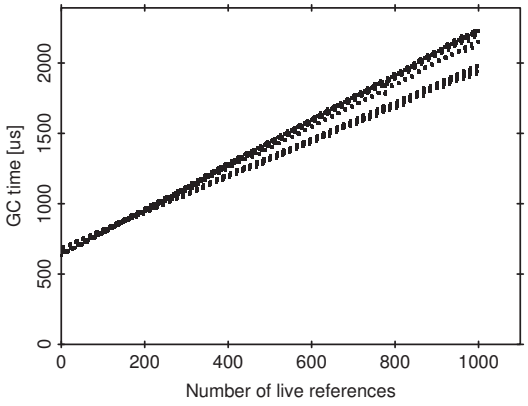

Figure 6: Measurement data of GC time as a function of number of live references.

Varying amount of references allocated We allocate one node with a constant size 1000, for which we vary the number of self references from 1 to 1000. The results of this is shown in Figure 8.

Varying number of nodes allocated We allocate a linked list where each node is of a constant size (16 words) and containing 1 reference (the next field). We vary the length of the list (i.e. the number of live nodes) from 1 to 1000. The results of this is shown in Figure 9.

From the measurements we can see that the behavior appears to be linear in all parameters. The scattered clustering of data points for some of the measurements can be explained by the, although relatively simple, non-flat behavior of the EMC. The reason

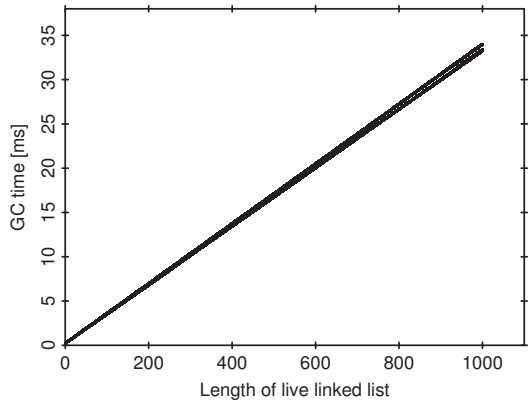

Figure 7: Measurement data of GC time as a function of the length of the live linked list.

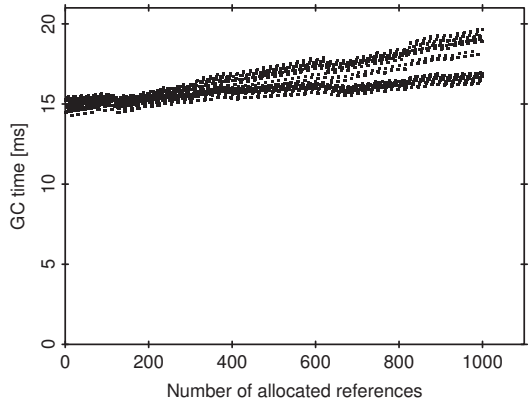

Figure 8: Measurement data of GC time as a function of number of references allocated by an interrupting task.

why, for some measurements, the clustering forms distinct lines instead of being more evenly distributed in an area is the discrete behavior of the EMC (i.e., either hit or miss in the pre-fetch cache) together with the way the heap parameters were controlled (keeping all but one constant). Nonetheless, the overall tendency is linear which supports the validity of the model.

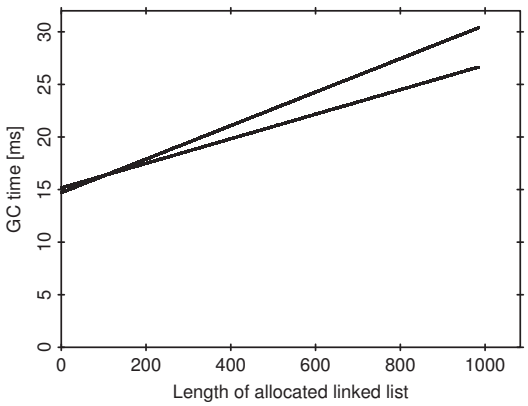

Figure 9: Measurement data of GC time as a function of the length of the linked list allocated by an interrupting task.

9 Related work

The key issue in scheduling concurrent garbage collection in real-time systems is undoubtedly how the garbage collector should be able to compete with the real-time tasks. Typically, this amounts to finding appropriate timing assumptions for the *collector task*. Although the garbage collector does not really have such timing properties, more or less artificial ones are necessary in order to determine schedulability of the whole system. Due to the quite complex dependency between properties of the real-time tasks and the required execution time for garbage collection, much of the focus in this field have unfortunately diverged from pure schedulability analysis towards improving measured performance.

In [12], Fu and Hauser presents a framework for describing a broad class of real-time garbage collectors and their corresponding scheduling premises. They also accurately identify the key to enable provable schedulability guarantees of a garbage collected system, to wit comprehensive knowledge about the memory behavior of the real-time tasks as well as the inter-dependencies between them and the garbage collector.

In [13], Henriksson presents a feasibility test for garbage collection based on *response time analysis* [16]. In contrast to our work, he assumes a per task parameter for worst-case garbage collection time required after one invocation (G_i). This parameter slightly corresponds to our execution time parameter due to new allocations (T_i^A). However, he does not present the connection between G_i and the actual garbage collection algorithm used. This work was later extended by Gestegård-Robertz and Henriksson to compute an upper-bound on the cycle time of the garbage collector in order to meet total heap memory limits [26], which corresponds to a rearrangement of our memory needs formula (Equation 4).

In [20], Kim et al. presents upper-bound estimates on the execution time of a garbage

collector based on Brook's [6] evacuation strategy. They present a schedulability test for the whole system based on a worst-case response time analysis of a sporadic server. In addition, they also present a live memory analysis to determine the worst-case local live heap memory of each task. In contrast to our work, they do not present a detailed connection between the parameters used in the execution time estimate and the actual garbage collection algorithm used. They are also limited to use rate monotonic priorities to enable the sporadic server schedulability test.

In [8], Chang presents a hybrid approach based on a lazy freeing reference counting collector and a backup mark-sweep collector. External fragmentation is avoided by using a fixed block size. He also presents a schedulability test based on a dual-priority scheduling scheme including the worst-case cost for both the reference counting collector and the mark-sweep collector. This is achieved by integrating the two garbage collectors into the real-time scheduling framework as tasks (i.e., derive appropriate timing assumptions for them). The main difference between his approach and ours is that he integrates the cost of garbage collection in the regular schedulability analysis. This makes it more difficult to extend the regular schedulability test with more features (e.g., shared resources) without affecting the schedulability test of the garbage collector.

Recently, Kalibera et al. developed schedulability tests for both time-based and slack-based scheduling of time-triggered garbage collection [17]. They show that none of them are superior to the other (in terms of schedulability) and they draw the conclusion that the choice of scheduling policy is a key part of designing garbage collected real-time systems.

10 Conclusion

We have shown how the reactive object model of Timber enables us to decouple the schedulability analysis of the real-time tasks from the cost of garbage collection. We have, based on the incremental copying garbage collector presented in [18] and the real-time programming language Timber [23], developed a feasibility test for the collector. We call this test *garbage collection demand analysis*, which contains only clearly identified parameters of the real-time system and their corresponding effect on garbage collection time. Apart from formal verification of correctness, we have confirmed the validity of the model through an experimental study run on the LPC2468 Developer's Kit from Embedded Artists.

The key motivation behind our approach to schedule garbage collection is to enable *any* scheduling policy (with corresponding feasibility test) for the real-time tasks. This is achieved by identifying and exploiting key properties of the run-time behavior of real-time tasks defined in Timber, which allow us to decouple the cost of garbage collection from the processor demand of the real-time tasks.

11 Further Work

Apart from timing assumptions and properties required by regular schedulability analyses (such as inter-arrival times, deadlines, worst-case execution times, etc.) our analysis also requires global live heap space bounds and heap allocation bounds for each task. Analyzing heap allocation properties for each task corresponds quite well to execution time analysis (with a slightly different cost model). Both are monotonically increasing accumulative properties. The results of such program analyses are typically expressed as functions of the programs input data [1]. In a real-time system, where tasks maintain an ongoing interaction with the environment, input data are typically tightly coupled with the state of the system. As a fortunate coincidence, such state-dependent properties corresponds very well to the behavior of global live heap space. Kero et al. has a forthcoming paper in preparation, presenting live heap space analysis for real-time systems [19].

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments.

References

[1] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Parametric inference of memory requirements for garbage collected languages. In *9th International Symposium on Memory management*, 2010.

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

[3] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *11th Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990.

[4] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, November 1990.

[5] A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science and Engineering, Oregon Health and Science University, April 2002.

[6] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256–262. ACM Press, August 1984.

[7] M. Carlsson, J. Nordlander, and D. Kieburtz. The Semantic Layers of Timber. In *The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. C Springer-Verlag.*, 2003.

[8] Y. Chang. *Garbage Collection for Flexible Hard Real-Time Systems.* PhD thesis, University of York, July 2007.

[9] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[10] 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):966–975, 1978.

[11] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, 1993.

[12] W. Fu and C. Hauser. A real-time garbage collection framework for embedded systems. In *Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 20–26, New York, NY, USA, 2005. ACM.

[13] R. Henriksson. Predictable Automatic Memory Management for Embedded Systems. In *OOPSLA'97 Workshop on Garbage Collection and Memory Management*, 1997.

[14] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems.* PhD thesis, Lund Institute of Technology, 1998.

[15] L. Huelsbergen and J. R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *Principles Practice of Parallel Programming*, pages 73–82, 1993.

[16] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[17] T. Kalibera, F. Pizlo, A. L. Hosking, and J. Vitek. Scheduling hard real-time garbage collection. *Real-Time Systems Symposium, IEEE International*, pages 81–92, 2009.

[18] M. Kero, J. Nordlander, and P. Lindgren. A Correct and Useful Incremental Copying Garbage Collector. In *Proceedings of the 2007 international symposium on Memory Management (ISMM'07)*, Montréal, Québec, Canada, October 2007.

[19] M. Kero, P. Pietrzak, and J. Nordlander. Live heap space bounds for real-time systems. In preparation. `http://staff.www.ltu.se/~keero/LiveHeap.pdf`, 2010.

[20] T. Kim, N. Chang, and H. Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software (JSS)*, 58(3):245–258, September 2001.

[21] C. M. Krishna and K. G. Shin. *Real-Time Systems*. McGraw-Hill, 1997.

[22] R. Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, 1999.

[23] J. Nordlander, M. Carlsson, A. Gill, P. Lindgren, and B. von Sydow. The Timber homepage. http://timber-lang.org, 2008.

[24] J. Nordlander, M. Carlsson, M. Jones, and J. Jonsson. Programming with Time-Constrained Reactions, 2005.

[25] J. Nordlander, M. P. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive Objects. In *Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, Arlington, VA, 2002.

[26] S. G. Robertz and R. Henriksson. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM.

[27] G. L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[28] Webpage. http://download.micron.com/pdf/datasheets/dram/ mobile/256MbSDRAMx32_low_power.pdf, April 2010.

[29] Webpage. http://embeddedartists.com/, April 2010.

[30] Webpage. http://www.standardics.nxp.com/products/lpc2000/ lpc24xx/, April 2010.