Roo: A Framework for Real-Time Threads

Chris Zimmermann Vinny Cahill

Distributed Systems Group,
Department of Computer Science,
Trinity College, Dublin 2, Ireland
{czimmerm, vjcahill}@dsg.cs.tcd.ie
http://www.dsg.cs.tcd.ie/dsg_people/{czimmerm/czimmerm.html, vjcahill/vjcahill.html}

Abstract

Traditional object-oriented real-time systems are often limited in that they provide only one approach to real-time object support. Taking the increasing demand for flexible and extensible object support environments into account, we discuss the design and implementation of a small object-oriented real-time executive based on a sub-framework which we call Roo. Roo is a component of the Tigger framework (our proposal for an extensible object support operating system) and is intended to support different object models providing soft real-time behaviour. Roo provides support for different mechanisms and policies for real-time thread management, scheduling and synchronization. In this it serves as a basis for other components of the Tigger framework.

1 Introduction

Traditional object-oriented real-time systems like [16, 11] are often limited in that they provide only one approach to real-time object support. As [21] points out, for programmers using these systems this sometimes presents a severe limitation: they are constrained by the mechanisms and policies provided by these systems and are unable to adapt them to their specific needs. To support such cases, an approach based on an extensible, object-oriented framework providing different mechanisms and policies for real-time support, from which the programmer is free to choose the most appropriate mechanisms, is required.

This paper discusses the design and implementation of a small sub-framework named Roo forming an executive for the support of soft real-time applications such as multimedia and interactive simulations. Developed as part of the Tigger project [7], this sub-framework will be used as a basis for the support of different active object models.

The remainder of the paper is structured as follows: after a brief discussion of the Tigger architecture including the ways in which active objects can be supported, Roo itself is described. First, the internal structure of Roo is described followed by the the building blocks from which Roo is composed. The following section outlines the benefits of this approach for the programmer using and extending Roo. A status report including some preliminary performance figures and an outlook conclude the paper.

2 Tigger Overview

This section gives a brief overview of the Tigger project and discusses our approach to supporting active objects and distributed real-time behaviour within Tigger.

2.1 Tigger

The Tigger project is developing a framework for the construction of a family—the Tigger Pride–of distributed object-support operating systems [7]. Members of the Tigger Pride will be hosted on top of bare hardware, (real-time) micro-kernels and conventional operating systems.

Each instantiation of the Tigger framework is intended to provide the necessary support for the use of some object-oriented language for the development of distributed and persistent applications thereby providing the basis for supporting different object models. Thus the fundamental interface provided by a Tigger is that provided for the language implementer and is a refinement of that of the Amadeus generic run-time library [6]. The interface used by an application developer is that provided by a supported language.

0-8186-7099-1/95 $4.00 © 1995 IEEE

The framework describes the fundamental abstractions supported by each instantiation and allows tailored implementations to be provided. Unusually, the Tigger framework is *self-hosting* thereby allowing instantiations to be built using distributed and persistent objects.

The baseline for the Tigger project is a set of minimal object-support operating systems supporting at least four primitive abstractions: distributed objects; persistent objects; activities (i.e. distributed threads of control) and extents (i.e. protected collections of objects). Members of the Tigger Pride may provide additional abstractions supporting, for example, security and transaction services. The result is that a Tigger which provides the necessary services for the target application domain can be constructed.

2.2 Concurrent Object Models

In order to provide maximum flexibility to the language designer, Tigger has to cater for different concurrent object models like asynchronous method invocation and object-bodies [26]. Therefore, Tigger supports a number of possible ways of associating threads and objects, allowing very fine-grained concurrency. In general, the way in which thread creation is associated with the creation of an object and the invocation of a method characterizes the object model. Fig. 1 summarizes the most important options. It should be realized that, in each case, whether or not a thread is actually created, may depend on factors such as the allowable degree of concurrency within a given object or other synchronization constraints.

Object Creation / Method Invocation	Do not Create Thread	Create Thread
Do not Create Thread	Passive Objects	Object Bodies
Create Thread	Reactive Objects	Autonomous Objects

Figure 1: Degrees of Concurrency Inside an Object

- Passive Objects: This is the traditional object model supported by object-oriented programming languages and support systems. During a method invocation, the thread of control active within the caller travels to the callee to carry out the invocation. New threads are created independently of object creation and method invocation.

- Reactive Objects: Each time a method invocation arrives at the callee, a new thread dedicated to this invocation is created. This models asynchronous invocation as discussed in [17]. A major advantage of this mechanism is that if the caller does not care about the return value of the method invocation, it can continue processing without having to wait for the callee to execute the method.

- Object Bodies: When an object is created, a dedicated thread of control is also created which is active inside the object during its lifetime [1]. This thread of control can be used for tasks not related to any particular method invocation such as checkpointing the state of an object while method invocations are carried out by the calling thread.

- Autonomous Objects: This variant combines features of the two previous models. Each time an object is instantiated, a thread is created to execute its body. In addition, each time a method invocation arrives, a new thread is created[1] for this invocation.

2.3 Supporting Distributed Real-Time Applications in Tigger

Based on the Tigger framework described above, Fig. 2 depicts our architecture for supporting application objects (AOs)[2] which require a distributed real-time environment.

This architecture is structured into several layers which are introduced briefly below. A more detailed description of this architecture can be found in [27].

The topmost layer offers a Toolbox Interface, from which higher, more application-oriented layers can choose certain object characteristics associated with

[1] Note that the creation of per-invocation threads can be done by the thread executing the object's body.

[2] In this case the term *application* stresses the fact that these objects are provided by higher layers and not by the Tigger framework itself. AOs can be anything from a true application-defined object such as an object handling multimedia data to an object implemented by a language implementor as discussed above.

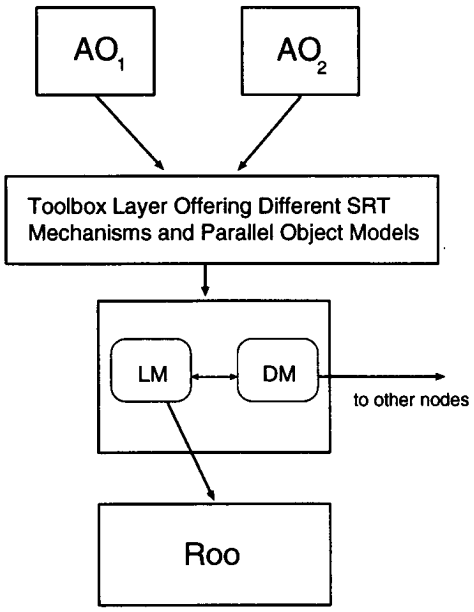

Figure 2: Overall System Architecture

real-time support like scheduling policies, synchronization mechanisms and the degree of concurrency inside an object specified by the number of active threads that can be attached to that object.

This interface allows very precise customization of the real-time behaviour of object execution, providing the programmer with a variety of mechanisms instead of constraining him or her to only one. This offers a flexible way of tailoring objects to specific needs.

The intermediate layer deals mainly with the management of the mechanisms offered by the Toolbox layer. It is subdivided into a distribution manager (DM), which interacts with DMs on other nodes of the distributed system in order to maintain real-time guarantees in the distributed case, and a local manager (LM) which controls the real-time behaviour of threads managed by the local real-time executive whose main responsibility is the provision of appropriate support for real-time threads, scheduling and synchronization as discussed in the remainder of this paper.

All layers including and below the Toolbox Interface define the soft real-time part (SRT) of the Tigger framework.

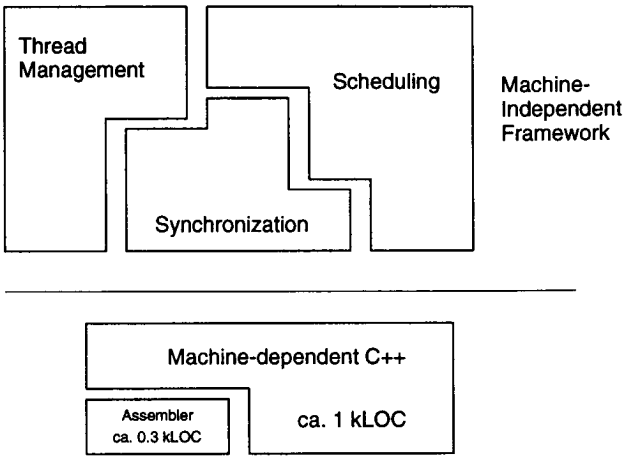

Figure 3: Software Architecture of Roo

3 Roo

Given the description of the overall architecture above, this section describes the real-time executive of Tigger—Roo.

3.1 Internal Structure of Roo

Since a major concern of Tigger is portability, Roo is divided into a machine-dependent part, which has to be kept to a minimum since it will have to be mostly rewritten whenever Roo is ported to a different hardware platform, and a machine-independent part, which forms the actual real-time support framework and resides on top of the machine-dependent part as depicted in Fig. 3.

This machine-dependent part is further subdivided into a small assembly language module, which contains functionality that cannot be expressed in ordinary C++[3] code such as context-switching and stack-manipulation, and a set of C++ modules offering a well-defined interface to the machine-independent part.

One could argue that the machine-dependent part as a whole could be implemented in assembly language in order to maximize efficiency and to gain additional speed-up. We decided to sacrifice these possible benefits in the hope that we could even reuse parts of this machine-dependent part when porting Roo to a different hardware platform. For this reason the functionality of the assembly language module is kept to a minimum and wherever possible we decided to use C++ instead of assembly language. Currently, this

[3] Our implementation language for Roo.

machine-dependent part of the i486 port of our architecture consists of approximately 1000 lines of well-documented source-code of which 300 lines are actually assembly language.

The real-time support framework is layered above this machine-dependent part and consists of three major building blocks providing thread management, scheduling and synchronization primitives. Each of these blocks is discussed in the following sections.

3.2 Thread Management

This building block provides an interface for the creation and destruction of threads as well as other services such as the ability to wait for a thread to terminate. As discussed above, Tigger supports different notions of active objects. By using this thread management interface, other components of Tigger can attach threads to an object at creation time as well as to individual invocations of object methods as appropriate.

From a conceptual point of view, the attachment of threads to individual objects resembles aggregation [5]. This is motivated by the fact that threads themselves are represented by objects and, apart from the passive object model, every object has at least one thread attached to it during its lifetime as discussed above.

Therefore, the functionality of the thread management building block is vital for the Toolbox layer as depicted in Fig. 2, since a major function of the Toolbox layer is the support of multiple object models.

3.3 Scheduling

Real-time scheduling is provided by the second major building block of the Roo framework as a class hierarchy. To achieve different real-time scheduling policies like Earliest Deadline First (EDF) or Rate Monotonic (RM) scheduling [18] within the framework a method called *scheduler stacking*[4] is employed which also enjoys the benefits of applying object-oriented techniques such as inheritance to achieve code-reuse.

Fig. 4 shows the overall structure of the scheduling class hierarchy. Here, an arrow denotes inheritance [4] and points to the derived class. Apart from simple inheritance of code, an arrow further stresses the fact a derived class like EDF makes strong use of methods supplied by its base classes.

The scheduler class hierarchy depicted works as

[4]The meaning of this term is motivated below.

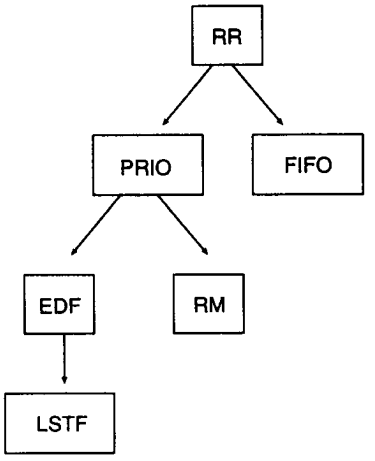

Figure 4: Inheritance Graph for Schedulers

follows[5]. The basis of the class hierarchy is a simple round robin scheduler (RR) which manages a circular list of runnable threads. This scheduler offers methods to place a thread at a certain position on the queue and to remove a thread from the queue. Preempting the currently running thread and selecting the next thread to be executed is not expensive; since the list of runnable threads is organized as a circular list, a simple pointer switch is all that is required.

A first-in-first-out scheduler (FIFO) is implemented on top of this RR scheduler: the only additional functionality that is required is removing the thread from the list when it releases the processor so that it is not scheduled again[6].

Entering the realm of priority-based scheduling, a scheduler capable of handling priorities (labeled PRIO in Fig. 4) uses the functionality provided by the RR scheduler to divide the circular list into N priority queues as depicted in Fig. 5 (smaller numbers represent higher priorities). In doing so, it overrides the methods for placing threads on and removing threads from the list provided by the RR scheduler. In addition to the thread itself, a priority must now be specified via the interface of the PRIO scheduler when placing a thread on the list.

The PRIO scheduler places the head of the first priority queue at the (logical) head of the circular

[5]In contrast to Fig. 4, which depicts the base/derived class relationship, it proves useful for the following discussion to imagine the drawing upside down with the box labelled RR at the bottom.

[6]This assumes that threads which are scheduled according to FIFO do not require to be executed periodically.

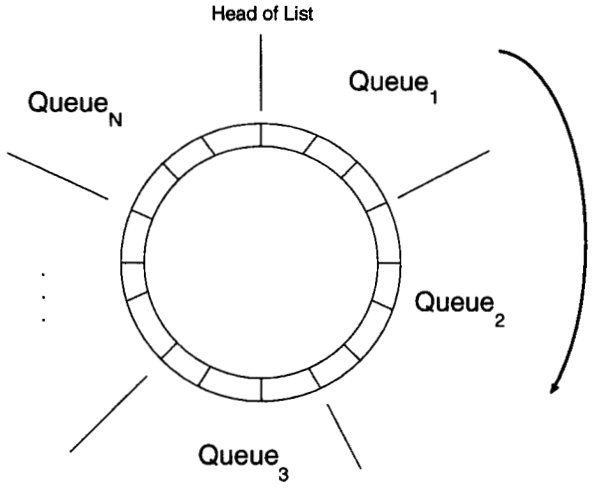

Figure 5: Mapping of PRIO onto RR

list managed by the RR scheduler. This ensures that threads with higher priorities are scheduled in favour of those with lower priorities.

Having laid these foundations, the remaining elements of the scheduler framework can now be discussed. A scheduler realizing Rate Monotonic scheduling (RM) is implemented easily using PRIO as a basis: each thread is assigned a static priority depending on its period [18] and then placed on the appropriate priority queue.

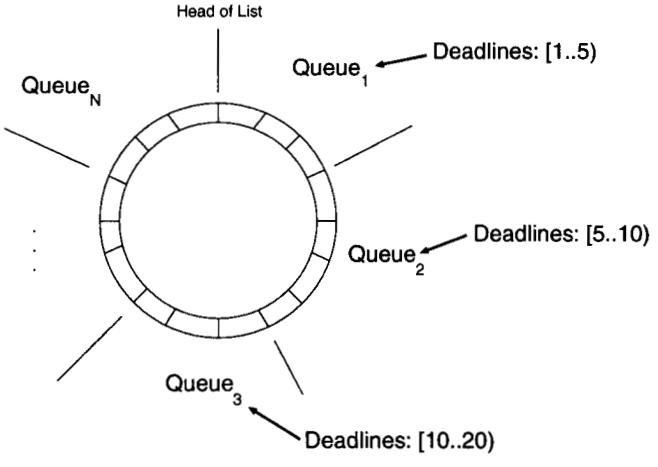

Figure 6: Mapping of EDF Scheduling onto PRIO

A scheduler handling the EDF policy is realized in the following way (see Fig. 6): possible deadlines are grouped into intervals and each interval is assigned a suitable priority according to its timeliness[7]. Threads on these priority queues are then ordered according to their deadlines, so that the thread with the earliest deadline in this interval is placed of its priority queue. Since Least Slack Time First (LSTF) is a refinement of the EDF algorithm [8], it is done is a similar way: each slack time interval is assigned a priority and threads are placed on the corresponding priority queue.

Since each of the classes makes strong use of its base classes, one can think of this structure as a stack: functionality is added by placing a scheduler realizing a different policy on top of an existing one. The scheduler implementing RR knows nothing about deadlines or priorities, and PRIO cannot handle slack time or deadlines, only the combination of classes achieves the functionality desired. Since complex scheduling algorithms are structured this way in Roo, the overhead added by each layer of this stack is comparably small.

The basic interface of the scheduler classes is depicted in Tab. 1. All scheduler classes derived from the baseclass RR must support this interface in order to be stackable. An exception is the method **move**, which assigns a new priority to a thread thereby moving it from one priority queue to different one and is defined in the PRIO class and below. Supporting this method in the baseclasses of PRIO does not make sense, since these scheduler classes do not know about priorities.

3.4 Synchronization Aspects

As depicted in Fig. 3, synchronization mechanisms represent the third major building bock of Roo which is also implemented as a class hierarchy. Similar to the scheduling class hierarchy described above, this hierarchy consists of a set of classes, implementing only basic functionality, from which higher-level abstractions with richer functionality can easily be built.

Take the realization of a monitor structure as an example of a general mutual exclusion (mutex) mechanism. According to [3], a monitor is a resource which can only be used exclusively. Before entering a monitor, a thread issues a `wait()`[8] call indicating that it is about to enter the monitor. If another thread is already using the monitor, the requesting thread will block until the other thread issues a `signal()` call indicating that it has left the monitor. This awakens

[7] An alternative would be to assign *each* deadline a different priority, but this would result in a possibly large number of sparsely populated priority queues, which makes the handling of these queues inefficient.

[8] A detailed discussion of the interfaces provided by the different synchronization classes is omitted due to spaces constrains.

Method Name	*Meaning*
`add(Thread)`	Add a thread to the scheduler realm
`remove(Thread)`	Remove a thread from the scheduler realm
`move(Thread, newPrio)`	Move a thread to new priority queue (`newPrio`)

Table 1: Basic Interface of the Scheduler Classes

the first thread which enters the monitor in turn.

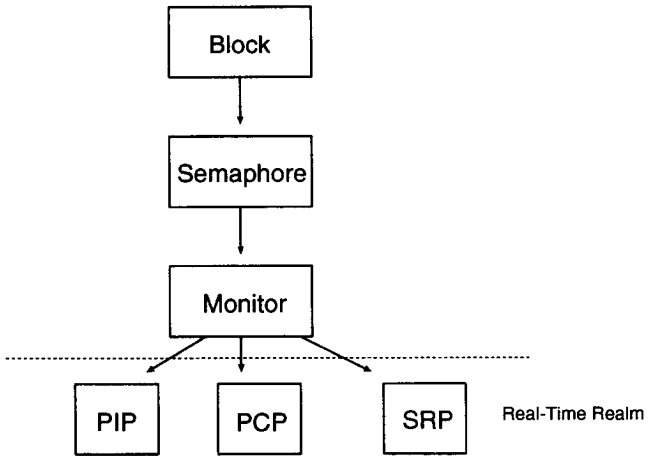

Figure 7: Inheritance Graph for Synchronization Mechanisms

The class structure involved in achieving this functionality is depicted in Fig. 7. The base class named BLOCK implements a basic blocking algorithm and offers only limited functionality: namely methods to block and release threads[9]. Using this restricted interface a SEMAPHORE class implements the usual functionality associated with semaphores [3]: managing a counter it blocks the calling thread when the counter reaches a negative value and releases the thread when the value is positive. The MONITOR class uses a binary semaphore to guard access to the monitor, directly using the interface and the code provided by the SEMAPHORE class.

As [20] points out, using this mutex mechanism on its own (i.e. without proper scheduling) can lead to the problem of priority inversion: a high-priority thread is blocked when trying to access a mutex which is currently held by a thread having a lower priority. A variety of protocols have been proposed to circumvent this problem [20, 2]. Since each of them has its advantages and disadvantages, Roo initially offers not only one but three protocols dealing with real-time constraints imposed on mutexes. These three protocols—namely the basic priority inheritance (PIP), priority ceiling (PCP) and stack reservation (SRP)—are implemented using the MONITOR class. Since the additional functionality compared to the original MUTEX class is restricted to scheduling issues, the overhead in terms of run-time performance imposed by using these derived classes is again comparably low.

[9]Since this base class is not intended to be used by other than sub-classes, its whole interface is `protected` in C++ terms meaning that no other classes than the ones inheriting from this base class can use its interface.

3.5 Discussion

The benefits of our approach are two-fold: the programmer using Roo[10] is offered maximum flexibility while the system programmer concerned with extending the functionality of Roo does so by extending existing class hierarchies.

3.5.1 Flexibility

As the above description of the features of Roo shows that it does not tie the programmer to a specific model but rather offers a variety of mechanisms, from which the programmer is free to select the most appropriate one. This approach offers maximum flexibility: whichever synchronization protocol or scheduling policy suits best can be selected. Therefore, Roo resembles a toolbox from which a programmer or even an object is free to choose the policy of its choice.

This approach is even more important when the desired real-time characteristics are not precisely known. An example from the area of distributed multimedia systems clarifies this point: consider an object displaying an MPEG video data stream [10]. Typically this object retrieves the data from a network connection or storage device, decompresses it and displays it in a window controlled by the windowing system. Various parameters like the frame rate in frames per second

[10]This would include the language implementor as discussed above.

or the resolution of the decompressed MPEG image directly influence the scheduling period, deadline and policy. Unfortunately, these *parameters* are normally not known in advance i.e. during the development. Using the toolbox provided by Roo, the MPEG object can influence its scheduling parameters and policy at run-time. For example, when decompressing a large picture, this object may choose to employ multiple threads concurrently to fulfill its task in order to speed up the decompression process by a finer granularity of parallelism.

3.5.2 Extensibility

From the viewpoint of the system programmer concerned with adding further functionality to Roo, this approach offers an easy way to extend the system. Since the basic mechanisms are already in place, they can be used without modification.

Take the scheduler stacking mechanism as an example. If the programmer wants to add another real-time scheduling algorithm, which we have not catered for in our original framework, he or she identifies the position in the scheduler stack at which to put the new scheduler class and inserts the new policy into the stack reusing code and functionality from the immediate baseclasses.

As an example, imagine the new scheduler uses priorities in order to implement a more advanced scheduling scheme where threads are moved between different priority queues depending on their run-time behaviour[11]. Since the basic PRIO class offers the method **move** taking a thread and its new priority as parameters and moving the thread from the priority queue it is currently on to the priority queue denoted by the new priority, all the programmer has to do is design the new class to decide, based on the given policy, when to call the **move** method of its baseclass PRIO.

3.5.3 Metalevel Issues

The flexibility is, from a conceptual point of view, achieved by the use of a metalevel architecture [15], where the characteristics of baselevel objects (the threads) are controlled[12] by so-called metaobjects [19], which in this case are the scheduler objects controlling the threads. When discussing metalevel architectures, the following issues arise [9]:

- *Upon what is reflected?* As the discussion above shows, the framework deals with control of the run-time behaviour of real-time threads. Therefore, the objects representing scheduling policies are the metaobjects of this model. Using the interface provided by the different scheduling objects (which is in turn the meta-interface for application objects), a programmer exerts control over the run-time behaviour of application objects indirectly by controlling the threads attached to the application objects.

- *What is the causal connection between the base- and the metalevel?* Recalling the description of the scheduler stacking mechanism from above, the foundation of the whole scheduler stack is the circular list managed by the lowest member of the stack: the RR scheduler class. By using the functionality supplied by this scheduler all other schedulers stacked above achieve their goal.

- *When does a level-shift between base- and meta-level happen?* Every time a new scheduling decision has to be made a level-shift takes place, and control is transferred from the baselevel to the metalevel. This is the case both when preemption occurs (during blocking, when a higher prioritized thread enters the scheduler framework) and when a thread exits.

3.6 Performance

This section gives some preliminary performance figures from our first prototype implementation. Table 2 gives the times for a context-switch and for making the scheduling decision using two different scheduling policies, namely round robin and priority-based. All figures were obtained using standard Intel-based PC hardware, in this case a 486DX2 running at 66 MHz. Note that context switching and the actual time the decision about which thread to dispatch next are separated; the former consists of a register-file switch only, whereas the latter includes up-calls to the various C++ routines which form the scheduler stack.

As shown in Tab. 2, the additional overhead of introducing priority-based scheduling is small compared to the overall-time it takes to make a scheduling decision[13] since the basic functionality is already provided by the round robin scheduler.

The poorer performance of Roo compared to other commercial real-time kernels such as QNX [12] can be

[11] A thread could be penalized by assigning it a lower priority if it uses up its whole time quantum for example.

[12] Or *reflected* upon, to use the conventional terminology.

[13] I.e. making the up-calls from the context-switching routine to the scheduler hierarchy.

Action	*Time in μsec*
Context Switch	3
Round Robin	105
Priority-Based	125

Table 2: Preliminary Performance Figures

accounted for by two factors: First, the current the implementation of Roo is merely a prototype and has not yet undergone the usual optimization process.

The second reason stems from our decision to emphasize portability as one of our design goals: for example instead of coding interrupt service routines which are responsible for scheduling decisions in assembly language and thereby hampering portability, we decided to implement the machine-dependent parts in C++, which impacts performance. Since it could be possible to implement the mechanisms discussed above in pure assembly language[14], the performance of highly-tuned, hand-crafted machine code could be achieved (sacrificing portability of course).

Considering our main target area[15], we think that a certain amount of performance can be sacrificed because we are not dealing with hard real-time systems where nearly every CPU cycle is crucial and failure to meet deadlines is disastrous.

4 Related work

Traditional object-oriented real-time systems like [16, 24] merely use the object-oriented paradigm as a means of structuring functionality and code, thereby providing the application with a more or less closed system. In contrast to this, our approach offers the programmer a variety of mechanisms and policies from which he or she is free to choose the most appropriate one.

The idea of using frameworks for operating system design is not new. Work such as [14, 13] makes heavy use of the object-oriented paradigm to structure, design and implement operating systems. Here, base classes provide architecture-independent control mechanisms such as virtual memory management, whereas concrete implementations use derived classes to take care of the platform-specific aspects like details of a specific memory management unit.

Also the use of metalevel architectures to design and implement operating systems or part thereof has been discussed in approaches like [25, 22]. In contrast to our work, these approaches provide only *one* mechanism thereby constraining the programmer to *one* policy, whereas our approach permits the use of configurable mechanisms yielding in application-specific policies.

5 Status and Outlook

A first prototype of Roo on a platform based on standard Intel-based PC hardware has been designed and implemented. Extending the Roo sub-framework with new schedulers and synchronization mechanisms is our next goal. We expect major input from the application areas where soft real-time is required: distributed multimedia systems and highly interactive, three-dimensional simulations. For example, decompressing and displaying a high-resolution MPEG video data stream may require a different scheduling policy compared to transmitting and processing a low-quality audio data stream.

After this, the next step will be the design and implementation of the Distribution Manager and the Local Manager. Since most of the functionality for implementing the LM is already provided by Roo, we do not expect major problems here. For the DM, work that has been done in the areas of distributed, object-oriented (hard) real-time systems such as [24, 11] serves as a starting point for future research. Again our framework approach should prove to be useful here.

6 Acknowledgements

We would like to thank Paul Taylor whose comments helped us to improve the quality of this paper and the anonymous referees for their suggestions.

References

[1] P. America. POOL-T: A Parallel Object-Oriented Language. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 199–220. MIT Press, 1987.

[2] T. P. Baker. Stack-Based Scheduling of Realtime Processes. *Real-Time Systems*, 3(1):67–99, 1991.

[14] With much effort only, because inheritance and other mechanisms offered by an object-oriented implementation language help much during the design and coding process.

[15] Soft real-time (mainly multimedia and interactive simulations).

[3] M. Ben-Ari. *Principles of Concurrent Programming.* Prentice Hall International, 1982.

[4] G. Blair et al., editors. *Object-oriented Languages, Systems and Applications.* Pitman, 1991.

[5] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, second edition, 1993.

[6] V. Cahill, S. Baker, G. Starovic, and C. Horn. Generic Runtime Support for Distributed Persistent Programming. *SIGPLAN Notices*, 28(10):144–161, 1993. Also technical report TCD-CS-93-37, Dept. of Computer Science, Trinity College Dublin.

[7] V. Cahill, C. Hogan, A. Judge, D. O'Grady, B. Tangney, and P. Taylor. Extensible Systems—The Tigger Approach. In *Proceedings of the SIGOPS European Workshop*, pages 151–153. ACM SIGOPS, Sept. 1994. Also technical report TCD-CS-94-45, Dept. of Computer Science, Trinity College Dublin.

[8] M. L. Dertouzos and A. K.-L. Mok. Multiprocessor On-Line Scheduling of Hard Real-Time Tasks. *IEEE Transactions on Software Engineering*, 15(12):1497–1506, 12 1989.

[9] J. Ferber. Computational Reflection in Class-Based Object-Oriented Languages. In *Proceedings of the 4^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, 1989.

[10] D. L. Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4):46–58, 1991.

[11] A. Gheith and K. Schwan. CHAOS^{arc}: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications. *ACM Trans. Comput. Syst.*, 11(1):31–71, 2 1993.

[12] D. Hildebrand. An Architectural Overview of QNX. In *Proceedings of the 1^{st} Usenix Workshop on Micro-Kernels and Other Kernel Architectures*, 1992.

[13] N. Islam and R. Campbell. Uniform Co-Scheduling Using Object-Oriented Design Techniques. In *International Conference on Decentralized and Distributed Systems*, Palma de Mallorca, Spain, 1993.

[14] R. E. Johnson and V. E. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS 91-1696, Department of Computer Science, University of Illinois, 1991.

[15] G. Kiczales et al. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[16] S.-T. Levi et al. The Maruti Hard Real-Time Operating System. *ACM Operating System Review*, 23(3):90–105, 1989.

[17] H. Lieberman. Concurrent Object-Oriented Programming in Act 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[18] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[19] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 2^{nd} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 147–155, 1987.

[20] L. Sha et al. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 9 1990.

[21] M. Staude. Planning Methods: Process Scheduling in Solaris 2.x (in German). *iX*, pages 130–136, 8 1994.

[22] K. Takashio and M. Tokoro. DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems. In *Proceedings of the 7^{th} Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 276–294, 1992.

[23] H. Tokuda et al. Real-Time Mach: Towards Predictable Real-Time Systems. In *Proceedings of the Usenix Mach Workshop*, 1990.

[24] H. Tokuda and C. W. Mercer. ARTS: A Distributed Real-Time Kernel. *ACM Operating System Review*, 23(3):29–52, 1989.

[25] Y. Yokote. Kernel Structuring for Object-Oriented Operating Systems: The Apertos Approach. In *Proceedings of the 1^{st} International Symposium on Object Technologies for Advanced Software*, pages 145–162. Springer Verlag, 1993.

[26] A. Yonezawa and M. Tokoro, editors. *Object-Oriented Concurrent Programming.* MIT Press, 1989.

[27] C. Zimmermann and V. Cahill. Raising the Cub. In *Proceedings of the Annual German Unix Users Group Conference*, pages 79–86, 1994.