A REAL-TIME OPERATING SYSTEM DESIGNED FOR PREDICTABILITY AND RUN-TIME SAFETY

Roberto Brega
Institute of Robotics, ETH Zürich, Switzerland, brega@ifr.mavt.ethz.ch

ABSTRACT

Complex embedded machines are usually controlled by an application software sitting on top of a real-time operating system. Thus, the safety issues involved do not apply only to the mechanical structure or to the electronic circuitry, but also to the software that is responsible for an efficient run-time and predictable error-handling. The single, least reliable, nonredundant piece building the system, will undermine the overall reliability of the whole machine.
This paper describes a real-time operating system (XOberon) with deterministic and safe run-time behavior, which is easy to program and to customize. XOberon goes beyond the established concepts, like interrupt- or priority-driven scheduling, bringing in new programming paradigms, such as deadline-driven scheduling, automatic storage reclamation, dynamic linking and loading, memory protection, and off-line/on-line duration computation.

INTRODUCTION

The principal responsibility of a real-time operating system (RTOS) can be summarized as that of producing correct results while meeting predefined deadlines in doing so. Therefore, the computational correctness of the system depends on both the logical correctness of the results it produces, and the timing correctness, i.e. the ability to meet the deadlines of its computation [1].
Hard real-time (HRT) operating systems can be thought as a particular subclass of RT systems, in which the lack of adherence to the above mentioned deadlines may result in a catastrophic system failure.
A RT application can be modeled as a set of cooperating tasks. These tasks can be classified according to their timing requirements, as hard real-time, and not real-time (NRT). A HRT task is a task whose timely (and logically correct) execution is labeled as critical to the operation of the whole system. The deadline associated to a HRT task is said to be a hard deadline. Consequently, it is assumed that missing a hard deadline can result in a system failure. NRT tasks are those tasks that exhibits no real-time requirements (e.g. system maintenance tasks running in the background).
Based on these observations we conclude that a RT operating system has to guarantee that each task can meet its timing requirements. However, it is worth noting that, in order to fulfill that responsibility, the objective of a RT operating system cannot be just that of minimizing the average response time of each application task; rather the fundamental concern of a RT operating system is that of being predictable [2, 3].

THE XOBERON/POWERPC RTOS

The XOberon/PowerPC hard real-time operating system is a rapid application development (RAD) tool for complex mechatronic products, developed at the Institute of Robotics (IfR), Swiss Federal Institute of Technology, Zürich (ETHZ), in order to address the aforementioned issues [4]. XOberon is loosely based on the *Oberon Operating System*, and it is written in the object-oriented programming language *Oberon-2*. Both the Oberon System and the programming language Oberon-2 were developed at the Institute for Computer Systems, ETH, Zürich, Switzerland. XOberon runs natively on VME boards based on the PowerPC microprocessor architecture [5]. Older versions of the operating system support Motorola 680x0 processors, too.
The core of XOberon is composed of: the real-time scheduler, the memory manager, the heap manager, the dynamic linking-loader, the communication support and the drivers toolbox. On top of them, several packages are available, like internet servers, board support packages,

the run-time performance monitor, etc. Hereafter the most important aspects of some of these components will be highlighted.

Scheduler

The real-time scheduler embodies the hard real-time nature of XOberon and defines the interface to the real-time system for use by the application programmer.

XOberon features a deadline-driven scheduler with admission testing, which presents to the application programmer an object-oriented abstraction layer for modeling user tasks. Each task fed to the scheduler must provide: a *run method*, an *exception-handler method*, a *duration* and a *deadline*, and, for repetitive tasks, a *period*. The duration is defined as the amount of foreground non–pre-empted processor-time needed to successfully complete and retire the longest execution path of the task's code. The time needed to fire the exception handler must also be taken into account, when specifying the duration. The deadline, which is part of the real-time problem definition, is the latest point in time, when logical correct results are considered usable. The period is defined as the time interval between schedules.

Upon allocation, each task is tested for admission by the scheduler. A positive answer, states that the operating system will be able to guarantee that the task being fed to the kernel, along with its exception handlers, will be started, executed and retired before the expiration of the deadline (i.e. the timing correctness is guaranteed), as long as the specified duration is correct. The scheduler constantly monitors the processing time being used by each task, and compares it to the allocated slot, in order to ensure that ill-behaved tasks cannot undermine the stability of well-behaved ones. Should the specified duration be violated, the operating system will trap the unfair task: the task will be blocked, removed from the scheduling pools, and a default exception handler will be fired. The same way, if for some reason, independent on duration violations, a deadline cannot be respected, the relevant task will be trapped with a deadline violation exception. A possible constellation yielding to such a condition, could be a real-time task that sleeps for some time (i.e. remains in the background, with no influence on its duration) and, while asleep, misses its deadline.

The process manager implements a static, shortest-deadline-first scheduling algorithm. The pool of real-time tasks is statically sorted according to the specified deadlines. The first one, i.e. the one with the shortest deadline, will be set for execution by the scheduler. This task will remain in the foreground, until its normal execution cycle completes, or when a task characterised by a shorter deadline is been activated by the occurrence of some event, like the expiration of a waiting period or the user intervention. Other, more dynamic algorithms (like shortest-slacktime-first) have also been taken into consideration, but they have been rejected (at the moment) because the higher overhead involved in the continuous sorting of the tasks pool doesn't pay back in a substantially higher processor utilisation [1].

The scheduler is also responsible for dispatching non–real-time tasks, hereafter referred to as threads. Since their calculations can be delivered anytime, threads are brought to the foreground only when no other real-time task is pending, waiting for being dispatched.

Threads have a priority associated with them; this value is used for deciding their dispatching order. The threads dispatcher, being pluggable and hot-swappable, can implement many scheduling algorithms: round-robin, top-sorting, priority-driven with priority aging, etc. The user can choose which scheduler is more appropriate for his application, or implement a custom-tailored one.

The scheduler incorporates the message passing and synchronization primitives. The model being implemented is congruent to the one present in the Java programming language: the Object class. This class, dubbed *Synchronizer* in the XOberon naming-scheme, fulfills the role of both an inter-process communication signal and a mutual exclusion primitive. Being an extensible class, new fields can be added to the standard ones, thus acting as a good synchronization and signaling bus between tasks. The mutual exclusion monitor is reentrant: a task holding a lock, can reenter the same critical section without incurring in a deadlock. The task scheduler is also responsible for tracking the correct use of the Synchronizers. Particular care has been devoted for avoiding unbalanced entering and exiting from a mutual exclusive region, and for releasing locked Synchronizers upon trapping of a task blocking some data-structures.

The scheduler also incorporates meta-information (information about information) for querying the various tasks' states, scheduling-lags, stack-frames, measured timings, etc. These capabilities are used by the outer-core modules, and they are available to the application programmer.

Memory Manager

The incorrect use of the available physical memory usually results in severe reliability problems in the operating system and in the applications. Micro-kernel based or Unix like operating systems typically solve this problem by implementing some memory-protection schemes. The main idea relies on using the available hardware support in order to let programs run in separate address spaces, communicating with each other via inter-process communication signals. This way, if an ill-behaved application has a memory-related fault (in the computer jargon, it crashes), only the offending application will be shutdown, without interfering with the operating system or with parallel running threads.

Unfortunately, allowing multiple address spaces presents some severe disadvantages, which are not tolerable on a real-time system. In fact, the overhead involved upon context-switch is huge; moreover very tight scheduling time-slices can contribute to the thrashing of the involved hardware resources, like data and instruction caches, translation-looakaside buffers and branch prediction tables [6].

Most of the memory related errors result from illegal pointer operations, such as noninitialized references to data structures, faulty pointer arithmetic, etc. Since the Oberon-2 programming language, like the Java programming language, is strong-typed and does not allow an explicit handling of pointer references, many problems, typical for the “c” programming language are caught at compile-time, and thus completely avoided. Nevertheless, there are some memory-related errors, which can be solved more efficiently during run-time, by using the paged memory management of modern processor architectures, like the PowerPC.

XOberon makes heavy and pervasive use of *paging*. The available physical memory is mapped, page by page, into the virtual address space (2^{32} bytes). Several, fixed-size chunks are assigned for the different uses. Building on this scheme, the Memory Manager reserves virtual blocks to the module loader, to the system-heap, to the stacks-pool, to the DMA range, and for the memory-mapped input/output. Each block is handled differently, according to its characteristics and is tuned for optimal cache-performance.

Using this memory management scheme, XOberon can handle the following memory-related conditions.

- *Nil-checks* (noninitialized pointer references) are resolved at run-time and not with code explicitly emitted by the compiler.
- A *Stack-overflow* can be safely trapped without compiler intervention or run-time overhead. Yet, stacks are allowed to dynamically grow (for non–real-time tasks) if there is such a need, hence allowing a minimal initial stack-size, with a consequently better memory utilisation.
- The *DMA ranges* can be marked as noncacheable, simplifying the handling of direct memory accesses for drivers writers.
- References to unloaded modules (dangling code or data pointers) can be easily trapped without having to risk an access on unknown memory blocks.

The proposed mechanism delivers a high degree of reliability and overall safety to the system. Moreover, the already minimal and real-time compatible run-time overhead is more than compensated by the speed increase gained by the optimal tuning of the cache behavior on the different memory blocks.

Heap Manager

Many programming languages allow the programmer to allocate and reclaim memory for data whose lifetimes are not determined by their lexical scope. Such data is said to be dynamically allocated. The area of memory where such objects live is called the *heap* [7].

In most systems, the management of heap memory is done by the application programmers, i.e. they must explicitly free heap memory at some point in the program by calling a *“free”* or *“dispose”* routine. Unfortunately, the manual reclamation of heap blocks is often unsatisfactory, because the lifetime of many data structures cannot be determined before execution. This is especially true for object-oriented languages where the goal of good design is achieved with the encapsulation of abstractions into objects that communicate through published interfaces. Hence, programmer-controlled storage management massively inhibits modular programming. Moreover, a lot of run-time errors usually hide inside missing or ill-placed disposes, the former resulting in memory leaks, the latter in dangling pointers. A safer solution is to let a system-wide daemon, namely a *Garbage Collector*, take care of the automatic reclamation of data, which is no longer referenced by running programs.

The implemented garbage collection scheme addresses the typical problems posed by the very high requirements of a real-time operating system.

First of all, given the real-time pre-emptive context-switch taking place in the scheduler, the garbage collector must ensure that high priority tasks can still be completed in a guaranteed amount of time, that is, without violating the specified time-constraints. This is done with an interruptible *Mark-and-Sweep* algorithm [8], by means of the collaboration between the garbage collector and the running tasks (referred in the literature as *mutator*).

A further important precondition to the safe garbage reclamation is the careful detection of pointers. Compile-time information, brought to the run-time system by the linking-loader, along with an heuristic algorithm [9] ensure that each pointer can be treated as such, and thus will not be left unseen.

The last issue is about being able of running in very tight memory conditions, i.e. when the amount of free memory tends towards zero. The frequently used *pointer-reversal algorithm* is memory efficient, but it is not usable in an incremental garbage collector that can be preempted by other processes; in fact, if the pointer-reversal is interrupted before the pointers are restored, the mutator cannot properly use these data structures. A more trivial algorithm for marking the live objects employs a recursive marking procedure. Unhappily, recursive procedure calls are not a practical method for marking data structures, being neither time- nor space-efficient, and may cause the system stack to overflow. XOberon improves performance and memory utilization by replacing recursive calls, iterating on an auxiliary stack, which is used to record the addresses of the heap-objects. The allocation procedure pushes the address of each dynamically allocated block on this stack. In this way, the garbage collector can be effective in abnormal memory conditions, still providing adequate performance.

The implemented garbage collector runs as a non–real-time task with variable priority: the less memory is available, the more prioritized will be the collector. Considering that the real-time tasks are inhibited by the compiler from allocating dynamic data (the allocation procedure being unbound in space and time), it is guaranteed that the garbage collector will always get adequate run-time for trying to reclaim unused data.

The proposed solution has proved to be very efficient in detecting nonreferenced objects and collecting them. It features high reliability and good performance, and contributes to increasing the overall run-time safety of the system. It is very well suited as the default garbage collector for a Java run-time, too.

Dynamic Linking and Loading

Modern software techniques require software components to have cleanly defined interfaces, in order to grant better inter-operability, ease of specification, and maintenance. Oberon programs are written as separate compilation unities called *modules*. Modules have an interface that is checked against violations during compile-time (for source code) and during linking-time (for object files).

In addition of boosting the overall safety and modularity of the run-time, compilation modules allow for a far shorter edit-compile-run cycles. The code emitted by the very fast, two-pass Oberon-2 compiler is transmitted through an ethernet link to the target machine, where the relatively small modules (usually less than 32 kilobytes in size) are loaded, then checked for version-consistency and eventually dynamically linked.

A linked module can be safely removed from the running system, if no other module is importing it. This is accomplished by a simple reference counting strategy. In the always possible case, where an implicit reference is held to some parts of the code or the data structures of a removed module, the memory manager blocks those dangling addresses, by trapping the ill-behaving task and firing a standard exception handler, as explained in the chapter *“Memory Manager”*.

The module loader contributes to an enhanced composability and customizing of the system, while ensuring a high degree of run-time safety and reliability.

Real-Time Code Analyzer/Profiler

The most difficult part, when defining a real-time task, is the definition of the duration value. Whereas the deadline is part of the real-time problem scope, the duration is dependent on the underlying hardware system, this being made up of a processor, a memory subsystem, and peripheral components [10, 11].

One possible solution could rely in the measurement of the task’s run-time with some system-wide clock, in order to have a reference duration value, which could then later be tuned. Of course, this is not a safe solution, since what is being measured is not the longest execution path, but a particular one. Moreover, a later intervention of some other processes could have dramatic influences in the cache performance, invalidating the information collected during development.

XOberon tries to solve this problem by providing a tight integration between the compiler and the run-time environment. The compiler profiles the longest path of a given task, by using the available information on the underlying system architecture. This information is fed to the target system, which continuously monitors and tunes the durations, by exploiting the PowerPC604 *Performance Monitor* [12, 13]. The Performance Monitor is a particular unit of the PowerPC604 microprocessor architecture, which constantly samples low-level, processor-related data, like the amount of L1-hits, the number of pipeline stalls, the frequency of mispredicted branches, and many other architectural events. This information is used for correcting the profiled durations fed to the run-time system towards a worst case, that more approaches real-world values.

Driver Support

The XOberon Driver Support follows two important concepts: the first one is about writing driver software, which exposes its functionality through published interfaces to an object-oriented database; the second aspect stresses the reliability and ease of implementation of a polling solution over plain interrupt handling.

The interface allows the on-the-fly configuration of peripheral components, like serial links, digital input and outputs, analog outputs, counters and so on. The user application will only need to reference driver objects – by name – in the database, decoupling the application from the used drivers. Consequently, a change in the hardware system will only need a reconfiguration of the drivers’ database, without requiring the application to be neither modified nor re-compiled.

The XOberon driver implementation guidelines suggest the use of polling over interrupts. Although the advantages of interrupts are clear (lower overhead, shorter response times), the polling solution is more deterministic, allows for guaranteed response times and it’s easier to write and maintain.

PERFORMANCE

Although raw speed was not one of the design goals, these being simplicity and safe run-time environment, the performance degree reached by XOberon/PowerPC is astounding. The scheduler, which is responsible for the hard deadlines, must stop a process, save its register-set, choose another candidate from the scheduling pools, and bring it in the foreground. The scheduler repeats these steps ten times every millisecond, or with a 10 kHz frequency. Such a scheduling frequency allows the programming of high performance controllers, which otherwise would have been hooked to an interrupt vector-entry, bringing complexity and taking determinism away, or by utilizing a custom developed hardware, which would not be easily maintainable or portable.

Scheduling Overhead

It is quite difficult to quantify the overhead brought by the scheduler, since it scales with the number of installed tasks. Anyway, after boot-strapping, the system overhead is less than one percent of the maximum available processor power on a MVME1600 board, with a PowerPC604 clocked at 100 MHz. At this point, the system is scheduling service-maintenance routines, along with communication drivers and internet daemons. The overhead scales linearly for more processes, with an increment of one percent in the worst case (this being a task that exhibits floating-point use with a 100 microseconds deadline).

As an example, a complex parallel machine (Hexaglide) with seven hard real-time, user-installed processes with a 300 microseconds deadline, and some non–real-time processes, brings the scheduling overhead close to five percent.

In the same way, the overhead decreases linearly with better PowerPC implementations, as on the MVME2600 board, powered by a 200 MHz PowerPC604e (Sirocco), or on the new MVME2300 board, with a 300 MHz PowerPC604ev (Mach5). In the first case, the scheduling overhead after boot-strapping drops to 0.4 percent.

How can XOberon achieve such a performance? The answer can be found in the actual scheduler implementation, which has been optimized by taking into account the deep knowledge the scheduler has about the running tasks, and the underlying processor architecture.

The scheduler has been optimized as follows:

- Hand-tuned assembler code, with branch prediction hints, careful cache tuning, instruction scheduling, stack pre-fetching, etc.
- Optimizations in the context-switching, on a per–process-base
- Fine-tuned memory management through paging
- Real-world, on-chip performance measurements for pipeline tuning

Instead of saving the full processor state upon task switch, XOberon uses the same object-oriented abstractions presented to the programmer, in order to choose an optimal way for saving (and restoring) this particular task's context. Because of this, some processes are context-switched faster than others, according to the process state and characteristics.

Floating-Point Performance

The raw floating-point performance of XOberon is remarkable, thanks to the fast PowerPC floating-point units. Their use has also being tuned for the most speculative execution mode. This computational power allows newer applications to be built, by calculating complex kinematics and dynamic models in real-time, with deadlines shorter than one millisecond, eventually allowing to direct control motors and actuators at their mechanical limits.

For example, the Hexaglide milling machine [14], is being run on a PowerPC604@100Mhz, with the following hard real-time tasks, amongst others (system maintenance or not relevant):

- PD-controller and velocity observer: period 300 microseconds, 130 fp-multiplies, and 120 fp-adds
- Path-planner: period 300 microseconds, 110 fp-multiplies, and 100 fp-adds
- Dynamic pre-controller: period 2.5 ms, 1720 fp-multiplies, and 1750 fp-adds
- Adaptation of dynamic parameters: period 10 ms, 400 fp-multiplies, and 380 fp-adds
- Data miner, watchdog, security processes.

The system is loaded up to the 89.8 percent by the hard real-time processes, with a 5.5 percent scheduling overhead.

The floating-point performance of the PowerPC processor architecture opens new opportunities for intelligent, efficient, simpler software-controlled mechatronic applications.

APPLICATIONS

XOberon/PowerPC is been successfully used in many mechatronic projects developed at the Institute of Robotics, ETH Zürich, Switzerland, and in other universities. *RoboJet*, *Hexaglide* and *MOPS* are the most representative.

Robojet [14] is a hydraulically actuated manipulator used in the tunneling construction work. Its task consists of spraying liquid concrete on the walls of new tunnels using a jet as its tool. Its application software implements an automatic and human oriented control system. This new tool enables the operator to manipulate the jet in various modes, from purely manual actuation of single joints to fully automated spraying of selected tunnel areas. The calculation of the redundant inverse kinemat-

ics and the closed-loop control of the eight hydraulic actuators is performed by the control system.
Hexaglide [15] is a six DOF parallel mechanism similar to the Stewart platform. It is the prototype of a new machine tool, which is being built at the Institute of Machine Tools and Manufacturing (IWF), together with the Institute of Robotics (IfR), ETH Zürich.
MOPS [16] is a service robot that picks up boxes with incoming mail at the ground floor of the IfR's five floor building, delivering them to the secretaries' offices, subsequently bringing back the outgoing mail to the ground floor station. Its navigation is based on the recognition of natural landmarks, which are compared to data of the building layout stored on the robot's processor.

CONCLUSIONS

The charter of XOberon is about providing a reliable, real-time capable run-time environment, with safety aspects guaranteed by the operating system. XOberon delivers to non–computer-scientists a valuable RAD tool for implementing embedded applications, which can then be deployed in high-demanding environments, requiring safety, determinism, and ease of maintenance. Nevertheless, the kernel schedules tasks with a 10 kHz rate, with an overhead of less than 1 percent on a PowerPC604 hardware. Moreover, the size of the complete XOberon operating system (core, network servers and clients) is less than 1 MB ROM and needs 1.5 MB RAM on the target.
The system prototype is currently being used as the software building block of many mechatronic projects developed at the Institute of Robotics, ETH Zürich, Switzerland, and in other universities.

ACKNOWLEDGEMENTS

I would like to thank the following people, who contributed with ideas, programming, or testing to the XOberon/PowerPC project: Prof. G. Schweitzer, Dr. S.J. Vestli, R. Hüppi, M. Honegger, G. Bianchi, M. Corti.

AVAILABILITY

Further information can be found at the Homepage of the Institute of Robotics: "*http://www.ifr.mavt.ethz.ch*", *Research — Real-Time Systems.*
The XOberon real-time operating system supports Motorola VME boards based on the 680x0 and PowerPC processor architectures. The development environment is based on Oberon System 3, and it is available for Windows95, WindowsNT and Solaris hosts.
XOberon/680x0 is available, free of charge, with source code, at "*ftp://clermont.ethz.ch/pub/XOberon/*".
XOberon/PowerPC is available for evaluation on a Windows CD-ROM.

REFERENCES

1. F. Panzieri, R. Davoli, Real Time Systems: A Tutorial, Laboratory for Computer Science, University of Bologna, Italy, Technical Report UBLCS-93-22, October 1993
2. J. A. Stankovic, K. Ramamritham, The Spring Kernel: A new paradigm for real-time systems, IEEE Software, Vol. 8, No. 3, May 1991
3. J. Mathai (edited by), Real-time Systems: Specification, Verification and Analysis, Prentice Hall International, 1996
4. D. Diez, S. J. Vestli, D'nia An Object Oriented Real-Time System, Real-Time Magazine, 95/3, pp. 51-54, March 1995
5. IBM, Motorola, PowerPC Microprocessor Family: The Programming Environments, 1997
6. J. L. Hennessy, D. A. Patterson, Computer Architecture: A Quantitative Approach, Second Edition, Morgan Kaufmann Publishers Inc., 1996
7. R. Jones, R. Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons Ltd., 1996
8. E. Dijkstra, On-the-fly Garbage Collection: An Exercise in Cooperation, Communications of the ACM, 1978
9. H.J. Böhm, Space Efficient Conservative Garbage Collection, ACM SIGPLAN PLDI 93, Albuquerque
10. P.G. Emma, Understanding some simple processor-performance limits, IBM Journal of Research and Development, Vol. 41, No. 3, 1997
11. N. Jouppi, David Wall, Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, Digital WRL Research Report 89/7
12. J. M. Anderson et Al, Continuous profiling: Where have all the cycles gone?, Proc. 16th SOSP, 1997, ACM SiGOPS
13. F.E. Levine, C. P. Roth, A programmer's view of performance monitoring in the PowerPC microprocessor, IBM Journal of Research and Development, Vol. 41, No. 3, 1997
14. M. Honegger, G. Schweitzer., O. Tschumi, F. Amberg, Vision Supported Operation of a Concrete Spraying Robot for Tunneling Work, Proc. M2VIP, Toowoomba Australia, 1997
15. M. Honegger, A. Codourey, E. Burdet, Adaptive Control of the Hexaglide, a 6 dof Parallel Manipulator, Proc. ICRA'97, Albuquerque USA, 1997
16. S.J. Vestli, N. Tschichold-Gürman, MOPS, A system for mail distribution in office type buildings, Service Robots Journal Vol. 2 No. 2, 1996