YASA - A Framework for Validation, Test, and Analysis of Real-time Scheduling Algorithms

Jan Blumenthal, Jens Hildebrandt, Frank Golatowski, Dirk Timmermann
Institute of Applied Microelectronics and Computer Science
University of Rostock
Richard-Wagner-Str. 31, 18069 Rostock, Germany
{jan.blumenthal, jens.hildebrandt, frank.golatowski, dirk.timmermann}@etechnik.uni-rostock.de

Abstract

This paper describes our work towards a rapid prototyping system for hard real-time systems focusing on scheduling algorithms and scheduler implementations. The framework aims to speed up the decision making process of a suitable scheduling algorithm for a real-time application. The framework supports various kinds of real-time scheduling algorithms, which can be simulated for evaluation purposes. Furthermore, implementations of these algorithms can be tested in a real-time operating system (RTOS) with real or synthetic workloads. The scheduler algorithms are implemented either as software routines, which are part of the operating system (OS), or are realized in a coprocessor to free the operating system kernel from time consuming scheduling operations. The target operating system of our framework is mainly RT-Linux, even though development for those systems is possible under Linux and Windows as well. The framework proposed in this paper is new in that it supports the selection of the best suitable scheduler based on real-time scheduling analysis.

1 Introduction

When designing an application that requires real-time capabilities of the underlying computer system, a real-time operating system is quite often used to relieve the developer from the burden of managing parallel execution of several tasks. Furthermore, usage of the operating system's application programming interface (API) makes a software product portable to different hardware environments and even to different operating systems using the same API. The downside is that very often, the developer has to achieve his or her goal not only due to the possibilities but also despite the limitations implicated by the OS. Especially in real-time applications where execution time is a strict constraint, these limitations cannot always be overcome by any possible workaround leading to the desired computational result.

One of these limitations is the scheduling algorithm implemented in a particular RTOS. Most commercial systems use static priority systems and only few of them offer alternative algorithms. The availability of open source RTOS, such as RTEMS or RT-Linux, gives the opportunity to improve existing scheduling methods in off-the-shelf real-time operating systems.

In recent years, some frameworks and tools have been developed based on schedulability analysis technologies. Such frameworks are PERTS [1], Real-time Graphic Analyzer RTGA [2], Stress [3], and Timewiz [4]. Furthermore, a reconfigurable scheduling framework for the RED-Linux Real-Time Kernel has been offered [5].

EVASCAN [6, 7] is a general framework that uses previous versions of YASA [8] as an application for schedulability analysis [9]. The completely redesigned framework YASA uses a more generic framework. It runs under both, Windows and Linux, with a simulator as well as with a real operating system as a target.

The extended framework YASA presented in this paper evaluates a task set with given properties in terms of schedulability and compliance with given execution time constraints while using different scheduling algorithms and scheduler imple-

mentations. Special attentions were given to dynamic scheduling methods and hardware supported scheduling. These technologies can provide very good performance but need thorough scheduling analysis to guarantee timeliness of the scheduled task set.
The remainder of this paper is organized as follows. In Section 2, important optimal scheduling algorithms are introduced and the technology of scheduling coprocessors is described. Section 3 explains the design and operation principle of YASA. In Section 4, special implementation issues of YASA and their consequences are discussed. Further on, this sections deals with the integration of different scheduling modules. Finally, Section 5 concludes the paper and gives some future perspectives of the YASA framework.

2 Basic Technologies

This section deals with the dynamic scheduling algorithms used inside the YASA framework and the supported scheduling coprocessors.

2.1 Scheduling Algorithms

Earliest Deadline First (EDF) introduced by Dertouzos [10] and Least Laxity First developed by Mok [11] have been proven to be optimal dynamic scheduling algorithms. A main advantage of these dynamic scheduling algorithms is a 100% processor utilization without deadline misses in theory. From the implementation point of view, it is easier to realize EDF than LLF (see Section 2.2.1). But both algorithms have some critical drawbacks, too. In an overload situation using EDF, all tasks of the given task set may miss their deadlines. This effect is known as *domino effect.* Another drawback of the EDF algorithm is the detection of a deadline miss only after it occurs. Whereas LLF is able to detect an impending deadline miss during execution of tasks. On the other hand, if two or more tasks have the smallest laxity at the same time LLF suffers from thrashing in evidence on an excessive number of context switches. The relatively high computational requirements are another downside of the LLF algorithm.
To overcome the *trashing effect,* we developed the Enhanced Least Laxity First algorithm ELLF [7]. This algorithm combines benefits from both algorithms, LLF and EDF. ELLF is an optimal scheduler. It benefits from processor utilizations of up to 100% and early detection of future missed deadlines. We propose a coprocessor as a hardware based scheduling accelerator to reduce the operating system' overhead caused by the periodic determination of laxities. This coprocessor supports RT-Linux as well as the YASA framework.

2.2 Task model for dynamic schedulers

A set of independent tasks τ (τ_i ... τ_m) is to be executed on a real-time system. Each task τ_i is characterized by several parameters described in Table 1. Based on these parameters, scheduling decisions have to be made.

Symbol	Description
$S(\tau)$	start time of task set τ
$A(\tau_i)$	arrival time of task τ_i
$C(\tau_i)$	computation time of task τ_i
$D(\tau_i)$	deadline of task τ_i
$P(\tau_i)$	period of task τ_i

TABLE 1: *Task Parameters*

2.2.1 Least Laxity First

For the Least Laxity First algorithm, the following parameters are important for the selection of the task to run next.

Symbol	Description
$L(\tau_i)$	laxity of task τ_i at start time
$L(\tau_i,t)$	laxity of task τ_i at time t
t	time since bootup, $0 < t < \infty$
$d(\tau_i,t)$	absolute deadline of task
$D(\tau_i,t)$	relative deadline of task τ_i in current period
$cd(\tau_i,t)$	consumed time in current period
$C(\tau_i,t)$	remaining computation time in current period

TABLE 2: *Timing Parameters*

The laxity of task τ_i is calculated by:

$$L(\tau_i) = D(\tau_i) - C(\tau_i) \quad (1)$$

The deadline of task τ_i at time t is:

$$D(\tau_i, t) = d(\tau_i, t) - t \quad (2)$$

The remaining computation time at time t is represented as follows:

$$C(\tau_i, t) = C(\tau_i) - cd(\tau_i, t) \quad (3)$$

The laxity at time t:

$$L(\tau_i, t) = D(\tau_i, t) - C(\tau_i, t) \quad (4)$$

Now, we substitute D and C:

$$L(\tau_i, t) = d(\tau_i, t) - t - (C(\tau_i) - cd(\tau_i, t)) \quad (5)$$

Based on equation (5), the scheduling decision in LLF algorithm is made. The task with the smallest laxity is selected to be executed next.

2.2.2 Earliest Deadline First

The EDF algorithm makes its scheduling decision based on equation (2). The task with the earliest deadline is selected for execution. The computation and comparison of deadlines is very simple and fast. The drawback of EDF is that in overload situations even tasks are executed which cannot finish until their deadlines. This consumes processing time that may be used to execute tasks that are still able to finish timely. Unfortunately, this malpractice can lead to miss deadlines of all remaining tasks.

2.2.3 Enhanced Least Laxity First

The ELLF algorithm is an improvement of the LLF algorithm. ELLF is more complex and makes its scheduling decisions in two steps. Since the detailed description of this algorithm is beyond the scope of this paper, only a short description of the operation principle is given here.
In the first step, as LLF algorithm, ELLF determines the tasks that have the smallest laxity. In the next step, out of these tasks the one with the earliest deadline is chosen and brought to execution while the other tasks enter a newly introduced task state. This state is called *excluded* state. It prevents from preempting the currently running task by elimating the trashing effect. This decreases the amount of unnecessary context switches, which makes the LLF algorithm practically unusable.

2.3 Scheduling Coprocessor

One method to speed up scheduling is to move the scheduler function as a whole or in parts to a dedicated hardware device. Of course, such a coprocessor is efficient only if the gain in execution speed of the implemented functions is not consumed by the time required to transfer both, input data and the computed results, between the hardware device and the operating system routines. Besides that, only functions that do not require access to CPU internal resources can be transferred to the coprocessor, i.e., registers. The scheduling parts that can be most efficiently implemented in hardware are online computation of priorities, priority comparison, and selection of the task to be executed next. The computation of priorities highly benefits from a hardware realization due to parallel calculation. Nevertheless, combination of all three parts of the scheduling operation in one hardware device avoids the necessity to transfer computed task priorities back to the operating system. Provided task states and task parameters used for priority computation are stored inside the coprocessor. The communication overhead at scheduling time merely consists of an optional start signal for the coprocessor and an identifier for the next task to run that is transmitted back to the operating system as the scheduling result.
Obviously, such hardware effort is justified only if the performance gain is significant. Hence, the type of scheduling coprocessor discussed here is used primarily for dynamic priority algorithms like EDF, LLF, or ELLF.

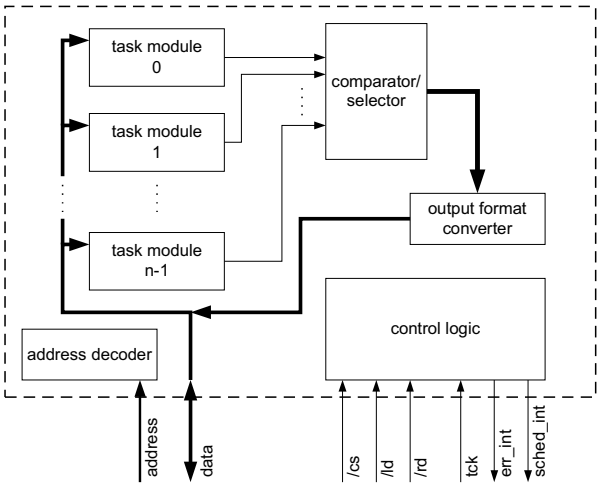

FIGURE 1: *Internal Design of the Scheduling Coprocessor*

The coprocessors used in the rapid prototyping framework have a design as given in Figure 1. Each task is represented by a functional block - the *task module.* It determines priority values and stores task parameters as well as task states. These values are accessible via registers that have to be initialized during system setup as well as if a new task starts. Task states have to be updated during runtime as they determine whether or not a task is active, i.e.,

it takes part in the scheduling process. The number and meaning of other parameters stored inside a task module depend on the implemented scheduling algorithm. For example, EDF algorithm uses task deadlines as scheduling criterion. Hence for EDF scheduling, task modules store the deadline in relative form, i.e., the number of remaining time units until the deadline is reached. This value is automatically counted down while the task is active. A missed deadline is equivalent to the deadline value being zero while the task is still active. More complex algorithms, such as LLF or ELLF, use - apart from the deadline - the remaining runtime of a task, a value that is automatically updated as long as a task is executed. Task laxity, the difference between remaining time until deadline and remaining runtime, is used to determine the order of task execution. These computations are done simultaneously for all tasks at scheduling time and thus make up the largest part in execution speed gain compared to a software solution.

Priority values computed inside the task modules are evaluated in the comparator module to determine the next task. The strategy used therein depends on the employed scheduling algorithm. The scheduling methods mentioned above determine the task with the smallest priority value. But there may be other algorithms that calculate the highest value or combine the comparison with some other conditions. In any case, the coprocessor returns an identifier that names the task belonging to that value determined by the comparator.

Although the coprocessors described here implement different algorithms, their interface to the host computer system and to the operating system is constant. This allows an easy integration of different hardware supported scheduling algorithms into the operating system. This means that different coprocessors can be compared with low overhead regarding their suitability for a given task set, since the surrounding operating system routines do not have to be significantly altered.

3 The YASA Framework

YASA is a new framework to develop, maintain, and evaluate hard real-time applications. It mainly considers the scheduling behavior in different environments. The YASA framework extends the operating system to the possibility to exchange the system scheduler at runtime. The supported schedulers use an API to a virtual scheduling environment of YASA. This API is necessary to make YASA platform independent. We call this virtual environment *Executive*. Further on, an Executive describes the target platform and encapsulates the original system scheduler. In YASA, applications are configured in projects. Each project consists of task definitions, scheduler definitions, Executives, and environment definitions. In the context of YASA, an environment is the configuration of the target system composed of an Executive, schedulers, tasks, and a cooperation of these modules, e.g., assigning schedulers to available processors. Applications in YASA are designed to run on different hardware platforms. They are only restricted due to Executives depending on special operating system functions.

Figure 2 shows the internal design of the YASA framework, which mainly consists of two components, the graphical user interface (GUI) based on the QT class library [12] and the command line based simulation environment (dotted rectangle). The whole developer project can be configured within the GUI. It is possible to define new tasks with properties, such as computation time, deadline, period, or the behavior in case of a deadline miss. For each task a start function can be defined. Furthermore, it is necessary to make some adjustments, such as selecting the desired Executive and selecting the scheduler for each CPU.

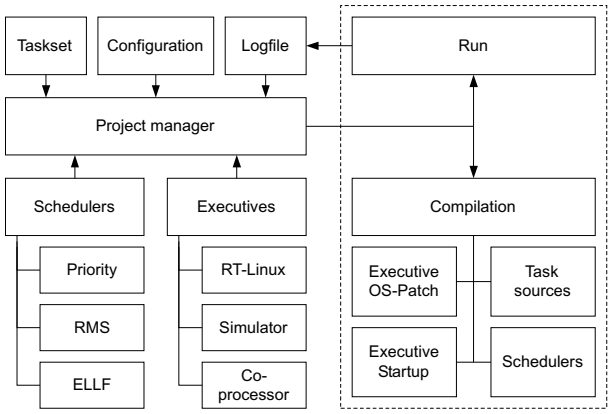

FIGURE 2: *Internal Design of the YASA Framework*

After entering all required information, the project can be compiled. The compilation process includes all selected components, such as Executives, schedulers, project functions, task information, and startup codes. These parts are linked together depending on the characteristics of the Executive. That can be a monolithic executable or a system of several shared libraries. In some cases, the kernel of the operating system has to be recompiled. After creating the executable, the project can be started. An internal log file mechanism registers all internal information such as the calling time of scheduler, deadlines, task switches, or locking of resources. After finishing execution, the log file is evaluated by

the graphical front end and is illustrated in clearly structured diagrams.
In the YASA framework, the different restrictions of particular operating systems result in tight limits. It is not be possible to run or compile certain task sets on every Executive, if these tasks use specific support functions of the underlying operating system. For instance, the function *malloc()* to allocate memory is a blocking function. For this reason, it is not available in kernel mode of most desktop operating systems in contrast to embedded systems. Task sets using this function cannot be successfully compiled on Executives running in kernel mode. The use of *kmalloc()* restricts the task sets to the maximum page size and to Executives running under Linux and its derivates.
Another restriction is given due to the implementation of the scheduling system. If the desired operating system supports only priority based scheduling, the selection of a scheduler is restricted.

4 Implementation Issues

4.1 Software Scheduler Modules

The software scheduler modules are defined in small ANSI-C modules. These modules, compiled as shared libraries or statically linked, are loaded and initialized at runtime during the startup phase of the Executive. Usually, the modules contain several member functions like *init()*, *schedule()*, or *cleanup()*.
In the initialization phase, a predefined attribute structure is used to configure the scheduler. Due to compatibility issues, the schedulers only use functions of the Executive to communicate with the environment. Information about the operating system and used hardware are hidden from the scheduler modules.
During runtime of the Executive, the member function *schedule()* of the current scheduler module is called every time the system scheduler is invoked to determine the task to be executed next. One result of the encapsulation process is that the actual scheduler function becomes very small. For instance, a priority based scheduler consists of just 25 lines of source code and is able to run on different Executives.

4.2 Hardware Scheduler Modules

Another feature of the framework, although still in the development phase, is the support of hardware scheduler modules which can be combined with special Executives. The scheduling coprocessors are realized in programmable hardware (FPGAs) that can be loaded with different configuration bit streams to implement different scheduling algorithms.
In YASA, the projects are independent from the scheduler type. Every task set can be scheduled with either software schedulers or hardware schedulers to compare the performance and trade performance gains against the effort of additional hardware.

4.3 The Executives

The Executive describes the environment in which the task sets and the desired schedulers are used. It provides an API to separate the schedulers from the underlying operating system. Access to different operating system structures and functions, such as threads and scheduling parameters is possible only via that interface. Furthermore, additional internal functions, such as making log files or synchronisation of task sets at start time, are included in the API.
In common operating systems, the Executive is integrated mostly by patching the original kernel. The final functional range of Executives depends mostly on features of the operating system as well as implementation details.
There are currently two completely different Executives the *Simulator* and the *RT-Linux Executive*.

4.3.1 Algorithm Simulation

The algorithm simulator is an Executive representing a virtual multi-processor machine. This Executive can be combined with all available schedulers defined in the framework. It supports different methods to handle deadline misses and is able to use different synchronization protocols to request resources, such as semaphores and mutexes. The simulator is independent from the operating system.
After the definition of a task set, the GUI generates source code containing a task array with attribute information about these tasks. Then, the compilation process is started with the translation of the Executive as a shared library or Windows DLL. Next, schedulers and task array are compiled to several objects and linked to one executable. The advantage of this spitting is that the Executive has to be recompiled only if the developer changes the Executives' settings. After compilation, the project is started and tasks are created based on information stored in the task array. All actions within the simulator can be logged during runtime and parsed in the GUI afterwards.
It is important to note that the simulator is not capable to simulate real-world programs. This Executive is ignoring start and end functions of tasks mentioned above. The reason is that different Executives are

running in different system environments. To run real-world programs within the simulator, it would be necessary to emulate the real world, in particular the API of the underlying operating system. Furthermore, interaction with the environment and the dynamic behavior of the application would have to be emulated. This is nearly impossible with reasonable simulation effort. To alleviate these effects, the framework provides methods to define start times of asynchronous tasks and allocation times of resources, such as mutexes. These times are used to simulate blockings and standby times. It is necessary to have in-depth knowledge about the progress of the tasks to get results close to reality. That is why the simulator is only useful to get information about general schedulability of the project.

4.3.2 RT-Linux (Integration in RT-Linux / API)

Real-Time Linux (RT-Linux) is a hard real-time extension to conventional Linux. The current version 3.1 is available on different hardware platforms, such as x86, PowerPC, and Alpha. It supports Single Processing as well as Symmetric Multiprocessing (SMP). We chose RT-Linux due to its POSIX style API, its possibility to change the original scheduler with small effort, and its open-source character based on GPL.

Usually, RT-Linux is started by loading several kernel modules like *rtl_sched.o* or *rtl_fifo.o*. The developer can change or improve these modules if necessary.

RT-Linux uses a static priority-based scheduler by default. The scheduling decision is based on the highest priority of one of the non-suspended tasks. The priority-based scheduler is very fast and simple but static. That means the priority value is independent from the scheduling process and can only be changed by calling functions of the POSIX-API. In the idle time of real-time kernel, if no real-time task is running, RT-Linux executes Linux as the lowest priority thread.

RT-Linux does not know anything about computation time, computed time, and deadlines. Thus, some enhancements were necessary to support dynamic schedulers. We simply added some variable definitions to the specific schedulers, threads, and scheduling structures to store the required information. In practice, the *Executive OS-Patch* of the RT-Linux Executive (Figure 2) is applied to original operating system. We did not change the current API of RT-Linux due to compatibility issues to conventional projects and future versions of RT-Linux. We patched some function bodies in the original scheduler module to guarantee the functionality of projects using default values.

Based on the original scheduler module of RT-Linux *rtl_sched.o*, an additional modular scheduler technology was implemented. This enables the exchange of schedulers during runtime for every CPU. In this way, the developer can run his or her own task set on different CPUs with different schedulers.

In contrast to the simulator Executive, the way to initialize a project in RT-Linux is slightly different. When the developer creates a new project, YASA generates the source code of the task set, basically an array of thread attributes. Next, YASA starts the compilation process of the required modules named *yp_project*, *ys_scheduler* and *logfilereader*. The first two modules are used to start the project and install the schedulers. The startup code of the RT-Linux Executive initializes all threads using the parameters stored within the task array. All threads are started in suspended mode with start time 0. After successful creation of all threads, a synchronized start follows and a reschedule is done.

To consider non real-time tasks within RT-Linux systems two possibilities exist. One method is to create non real-time tasks as Linux processes. Unfortunately, the communication between RT-Linux threads and non real-time Linux processes is inefficient and unpredictable due to extra scheduling within the Linux-Kernel and non-blocking communication between the two kernels. The other method is the integration as RT-Linux threads. Therefore, it is necessary to initialize the task structures with default parameters guaranteeing that these tasks are executed only when there is no real-time task pending.

During runtime, the Linux thread may run in the idle time of the real-time kernel. Hence, in fully utilized real-time environments it may seem like a hanging Linux system. Therefore, YASA has an mechanism to stop projects. After reaching the defined end time of a task set, the project is stopped. Finally, the *logfilereader* module is started to transfer the scheduling information into user space (Linux).

Most of the schedulers used in YASA require detailed information about task properties such as computation time. The developer should be aware that it can be a lengthy and intricate task to determine this necessary values. The determination of these values must be done before entering the periodic state of the threads during runtime. The computation time for instance is depending on everything in the computer: processor speed, memory performance, I/O activities, caches, and of course the system environment such as interrupts caused by hardware events. If the pre-calculated computation time of a task is smaller than the final execution time, many of the

dynamic schedulers do not work correctly.

4.3.3 RT-Linux Executive with HW-Scheduler

The RT-Linux Executive with hardware scheduler support is an improvement of the RT-Linux Executive. It is similar to the Executive explained in Section 4.3.2 but it uses a hardware scheduler to speed up the scheduling process. Task sets running successful with the RT-Linux Executive run with this Executive, too.
At start time of a project, the coprocessor device is loaded with a configuration according to the selected scheduling algorithm. Moreover, it is initialized with the task parameters of the project. Changes of task parameters at runtime have to be propagated to the coprocessor. Every time the system scheduler is called, the Executive starts the coprocessor to determine the task to be executed next.

5 Conclusions

The YASA framework presented in this paper enables the development of real-time applications for different operating systems. It gives the developer the possibility to evaluate the schedulability of his or her application from an early simulation with synthetic load to execution of real code using hardware or software schedulers.
The GUI of the framework allows easy comparison of scheduling methods and technologies by giving an overview over scheduling events, resource requirements, or execution times. This predestines YASA as an excellent program for education purposes as well.
Future developments of YASA see the extension of the range of supported scheduling algorithms and Executives. Furthermore, more comprehensive integration of scheduling coprocessors is focused.

References

[1] J.W.S. Liu, C. L. Liu, Z. Deng, T.S. Tia, J.Sun, M. Storch, D. Hull, J. L. Redondo, R. Bettati, A. Silberman: "PERTS: A prototyping environment for real-time systems", International Journal of Software Engineering and Knowledge Engineering, 6(2):161-177, 1996.

[2] I. Ripoll: "RTGA Real-time Graphic Analyzer", http://bernia.disca.upv.es/rtportal/apps/rtga/index.html Valencia, Spain, 2002

[3] N.C. Audsley, A. Burns, M.F. Richardson, A.J. Wellings: "Stress: A Simulator for Hard Real-Time Systems", Software-Practice and Experience, Vol. 24(6), p. 543, 564 (June 1994).

[4] "Timewiz - An Architectural modelling, analysis, and simulation environment for real-time systems", White paper, http://www.timesys.com/pdf/timewiz_ds.pdf

[5] Y-C Wang, K-L Lin: "Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel" 20^{th} IEEE Real-Time Systems Symposium Phoenix, Arizona, 1999

[6] F. Golatowski, J. Hildebrandt, D. Timmermann: "Rapid Prototyping with Reconfigurable Hardware for Embedded Hard Real-Time Systems", 19^{th} IEEE Real-Time Systems Symposium 98 WIP - Session, Madrid, Spain, 1998

[7] F. Golatowski, J. Hildebrandt, D. Timmermann: "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems", 11^{th} Euromicro Conference on Real-Time Systems 1999 York, England, 1999

[8] M. Schlegel, F. Golatowski: "Yasa - Yet Another Schedulability Anayzer", http://www-md.e-technik.uni-rostock.de/ma/gol/yasa/ University of Rostock, 1998

[9] F. Golatowski, D. Timmermann: "Using Hartstone Uniprocessor Benchmark in a Real-Time Systems Course", 3^{th} IEEE Real-Time Systems Education Workshop Poznan, Poland, 1998

[10] M. Dertouzos, Control Robotics: "The procedural control of physical processors", IFIP Congress, pp 807-813 1974

[11] A.K. Mok: "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment", Ph.D. Dissertation, MIT, 1983

[12] Trolltech Inc: "QT class library", http://www.trolltech.com, Oslo/Norway, 2003