DCOS, A REAL-TIME LIGHT-WEIGHT DATA CENTRIC OPERATING SYSTEM

Tjerk J. Hofmeijer and Stefan O. Dullman and Pierre G. Jansen and Paul J. Havinga
Computer Science
University of Twente
Enschede, The Netherlands
email: {hofmeijer, dulman, jansen, havinga}@ewi.utwente.nl

ABSTRACT
DCOS is a Data Centric lightweight Operating System for embedded devices. Despite limited energy and hardware resources, it supports a data driven architecture with provisions for dynamic loadable Modules. It combines these with Real-Time provisions based on Earliest Deadline First with a simple but smart resource handling mechanism. We will give an overview of the capabilities of DCOS and we will describe the basics of the main mechanisms[1].

KEY WORDS
Embedded, distributed, low-energy, resource handling

1 Introduction

Nowadays embedded micro-controllers are becoming smaller and more energy efficient. That enables the design of tiny, energy efficient embedded devices that shrink in size but grow in functionality. This increase in complexity leads to a different approach for software design. Programming firmware as, for example, a classic super-loop implementation becomes too difficult to design and maintain, and is inefficient concerning processing power and energy consumption.

To simplify software development for these new devices, small operating systems are developed, targeting these tiny micro controllers. This paper introduces such an operating system: the Data Centric Operating System (DCOS).

1.1 DCOS Overview

DCOS is a *Real-Time Operating System* (RTOS) for embedded devices with very limited memory, processing, and energy resources. Despite these limitations, DCOS has powerful features like, real-time scheduling, online reconfiguration, and support for a modular data driven architecture.

Where other operating systems for tiny embedded applications offer configuration only during compile time, DCOS is a dynamic system, able to adapt its functionality to create the most efficient configuration for every situation.

Furthermore, with the module support in DCOS, applications can be defined as modules, compiled off-line and dynamically inserted, or removed, in binary format. Firmware can now be upgraded by replacing only certain parts, instead of the complete binary. This simplifies the upgrade process, and it limits the use of precious energy.

In what way differs DCOS from other systems? To answer that question we describe the unique aspects of DCOS as we have not seen in comparable systems in the remainder of this section.

Real-Time EDF scheduling

DCOS uses real-time preemptive EDF scheduling [12] on hardware considered of having too limited resources to do so. In order to simplify scheduling, other systems use methods as (1) cooperative scheduling [10], where the real-time behaviour is mostly the responsibility of the programmer, or (2) event driven operations like TinyOS [6], where system behaviour is unpredictable and real-time guarantees can not be given. By using these methods, the other systems rule out the advantages of preemptive scheduling, like a better responsiveness and provisions for Quality of Service of streaming media. The organisation of real-time processing is given in detail in section 4.

Automatic mutual exclusion

The kernel enforces mutual exclusion of shared resources without the need of semaphores or monitors. By adding to a task a list of used resources, the scheduler uses a smart but simple mechanism to determine if there are tasks that share resources and schedules these tasks so that concurrent access to these resources is excluded. This simplifies application development in contrast to other systems where the application programmer must perform the error prone task of providing these mechanisms himself. The organisation of the automatic mutual exclusion is explained in section 4.

[1]This research is partly sponsored by the EU project Eyes, IST 2001-34734 and by the IBM Equinox Programme.

Data centric architecture

DCOS supports the data centric architecture [5][11]. With the data centric architecture the components of an embedded applications can be enabled or disabled, or mutually rearranged. Connections between the different components are data streams that are centrally coordinated. This enables reconfiguration of functionality online, instead at compilation time only. The support for the data centric architecture provides functionality for *Inter Process Communication* (IPC) and hence no separate mechanism is needed.

1.2 Platform

A prototype of DCOS is implemented on a Texas Instruments MSP430 micro-controller, running at 4.6MHz, with 2048 bytes of *Random Access Memory* (RAM), and 60KB of program flash memory. The kernel size is 3800 bytes, and the kernel itself needs 32 bytes of RAM.

2 Related work

Recent years have shown a growing interest in developing systems and applications for very-low-cost embedded systems with severely limited ROM and RAM. A typical example of a development environment for sensor networks is TinyOS. TinyOS is an open-source component-based architecture designed for wireless embedded sensor networks. Even though it does not comply to a strict definition of operating system, its component library includes various network protocols, distributed services, sensor drivers, and data acquisition.

Salvo is a Real-Time Operating System (RTOS) designed expressly for very-low-cost embedded systems with severely limited ROM and RAM. Typical applications use 1-2K ROM and 50-100 bytes of RAM. Salvo is highly configurable and scalable, with a full set of run-time features including priority-based, cooperative multitasking, event services, real-time delays and elapsed-time services.

The real time scheduling techniques presented in this paper are based on the combination of EDF and inheritance techniques as presented in [7] for use of shared resources during the complete run-time in tasks and in [8] in which the use of shared resources is organised within nested NCSs. The integration of EDF and inheritance are integrated in a novel way to constitute an attractive set of scheduling and dispatching techniques with a straightforward computation of blocking techniques. We will use these techniques in the context of this paper.

The foundation for these techniques are based on EDF [12] with the use of shared resources [9] and the inheritance techniques are based on the work of Baker [1] and Sha [15].

The scheduling algorithm is similar to Baker's Stack Resource (SR) protocol [1]. As in SR we use Critical Sections, originally introduced by Dijkstra [4] and profitably used by Sha in [15] to confine the problematic unrestricted use of shared resources. However, SR is a multi-unit protocol with which the computation of blocking is an (NP-)hard problem. EDFI can be considered as a simplified version of SR with single unit resources for which a straightforward feasibility analysis can be shown. For this limited model we can use a priority inheritance technique, which is similar to the Priority Ceiling (PC) protocol of Sha et al. [14]. However, PC uses fixed priorities for inheritance while EDFI uses Deadline Inheritance (DI). DI allows for a considerable simplification of blocking computation during feasibility analysis.

EDF based systems with shared resources have been investigated earlier by Jeffay [9]. However, Jeffay does not base the use of shared resources on NCSs. Instead he partitions a task into *phases* in which it is allowed to use *one* resource only. Scheduling is executed according to EDF with Dynamic Deadline Modification (DDM). DDM is a technique based on the dynamic introduction of *execution deadlines*, which prevents the preemption of a running task by another – shorter deadline – task if both tasks share mutually exclusive resources. These execution deadlines are determined dynamically, at the cost of the real-time budget.

The difference between DDM and our approach is that DDM uses phases with *dynamic* deadline modification, while EDFI uses NCSs with *static* deadline inheritance. We estimate that the real-time computation cost of DDM is considerably higher than our approach which requires very few work to be done at real-time budget costs.

The principle behind scheduling a task set with shared resources is that a released task τ_i stays on the Released Queue as long as it needs resources that are already in use by one of the tasks in the Run Stack, even if τ_i has a shorter deadline. Therefore, once a task τ_t is on the Run Stack, it will never claim a resource already held by another, preempted, task. Such a task τ_t would simply not have been scheduled. We enforce this behaviour by *deadline inheritance*, which is similar to Priority Inheritance, introduced by Sha [14].

3 The DCOS Kernel

This section gives a short overview of the functionality available in the DCOS kernel.

3.1 Real-Time Scheduler

DCOS has an RT-Transactions EDFI scheduler [7, 8] based on *Earliest Deadline First* and deadline *Inheritance*. EDFI is a lightweight preemptive hard real-time scheduling algorithm. Mutual exclusion of shared is resources, based on deadline Inheritance techniques, is enforced by the scheduler itself. Through analysis of a given task-set a hard guarantee on meeting the real-time constraints for each task can be given. More details of the scheduler are given in section 4.

3.2 Data Manager

The kernel supports a data centric architecture. Such an architecture enables the application to dynamically reconfigure its functionality. The main differences of the data centric architecture to a static configured application is that the functional building blocks are centrally coordinated, and that these blocks are loosely coupled (meaning that a block has no hard coded connections to other blocks). Rearranging the connections will create different configurations, making the system functionally adaptable.

In the data centric architecture, data is associated with its processing. The processing entities are called *Data Centric Entities* (DCE) while the relevant data is referenced by a so called *Data Type* (DT). A DT is a data object that can be read, written and signalled by a DCE. A publish/subscribe mechanism is used publish DTs to DCEs. When DT has been changed the subscribed DCEs are signalled. For instance an event, which is an interrupt or a signal, can signal a DT that will cause the subscribed DCEs to be activated. Note that in DCOS the signal handling is coordinated by a so-called data-manager.

New configurations can be achieved by altering the set of active DCEs and modifying their subscriptions. A similar but much more complex use of such a mechanism has been shown in the Splice system [2]. In Splice DTs are called "data-sorts".

3.3 Dynamic Loadable Modules

DCEs can be implemented as loadable modules that can handle DTs. DCOS is able to support reconfiguration of the system based on the inclusion or removal of modules. These modules are are not part of the operating system itself but provide additional application or system functionality. Modules can be loaded dynamically, that is at runtime. Note that this may require admission control to guarantee real-time behaviour.

A *Dynamic Loadable Module* (DLM) is a task compiled separately from the kernel code. A DLM is relocatable and can be loaded and executed anywhere in program memory, and as such it is a building block for the creating of new configurations online. With this module support, modifications of applications can be done more efficiently. Instead of updating the complete application (such as described in [13]) only a subset of the modules making up the task-set has to be changed, resulting in less data traffic and thus less energy consumption. Another advantage is that it allows nodes in a network to be heterogeneous, each node may execute its dedicated set of modules and as result less memory space is needed.

A ready DLM may be transferred to the target hardware through the radio or the serial port where on arrival it is stored in the secondary storage from where it can be loaded for execution. For the communication a packet protocol is provided. This protocol divides the DLM into small sub-packets which are uploaded individually so that the node can store it temporary in RAM before writing it to the secondary storage. This protocol can be used for any type of binary that has to be uploaded to a file system on secondary storage (EEPROM). This is a low complexity file-system that only supports location, creation and reading of files.

3.4 Other support

DCOS is able to dynamically reconfigure itself by altering its task - and data sets. So DCOS supports dynamic memory allocation. Furthermore, in the implementation of DCOS we have support for various kinds of devices such as the radio transceiver, a serial port, a serial EEPROM with file system, as well as an LCD. The details of these implementations are beyond the scope of this article.

4 Real-time operation

This section describes the real-time concepts and operations as used in DCOS. DCOS uses lightweight RT scheduling and dispatching based on *preemptive* Earliest Deadline First (EDF). Shared resources can be used under mutual exclusion. This, in general, may complicate scheduling, resource synchronisation and switching and may confront the application programmer with a rather complicated environment. However the combination of EDF with Deadline Inheritance (DI) to EDFI, as described in [7, 8], the mutual exclusive use of resources is provided elegantly by the scheduler.

EDFI limits process switching, while mutual exclusion of shared resources is granted at system level so that the programmer does not need to take care of resource synchronisation: processes are simply not scheduled by the system whenever there is the threat of a potential resource conflict.

EDFI can manage scheduling and dispatching very efficiently. It uses few system code and processing overhead and it hardly needs additional memory (RAM). Therefore, it is suitable for lightweight micro kernels. Therefore we will use EDFI for DCOS.

4.1 Real-time task specification

A task in DCOS is a real-time task. The kernel may run a varying number of these real-time tasks, each of which could have been inserted dynamically as a module. The set of tasks that may request attention of the processor is called the *task set*. Every task τ_i with $(1 \leq i \leq n)$ is defined by a set of properties which are relevant for it timely behaviour. These properties, which must be provided by the application designer, are described as follows:

- *Deadline Interval* D_i is the relative time between the arrival a_i of a task and its absolute deadline d_i, where $D_i = d_i - a_i$. A task must be finished before its absolute deadline otherwise a fault has occurred.

- *Period* T_i is the minimum time between every invocation of a task. Note that tasks may be aperiodic, but the inter-arrival time between two successive invocations of τ_i must not be smaller than T_i.

- *CPU Cost* C_i is the worst-case computation time of a task i, denoted as C_i.

- *Resource Usage* $\mathcal{R}_i$ is a list of resources, also shared by other tasks, that τ_i will use during each invocation. A task may *not* hold resources between invocations.

A complete real-time specification of a task looks as follows:

$$\tau_i : (C_i, T_i, D_i, \mathcal{R}_i) \qquad (1)$$

which is exactly what an application programmer has to specify. Nothing more and nothing less. The system can do the additional work: based on the task set specification DCOS can execute the feasibility analysis for the admission control of tasks and/or modules. If the addition of a task/module implies a non-feasible set of tasks, then the task is rejected; otherwise it is inserted in the system. The feasibility analysis is beyond the scope of this paper and for this we refer to [7, 8].

We will describe shared resource and its consequences for task synchronisation in section 4.5.

4.2 Real-Time multitasking

EDFI uses dynamic priorities determined by the absolute deadlines. Such a deadline is easily determined by adding D_i to the arrival time of the task's invocation. The arrival time is determined by a periodic clock, by an aperiodic interrupt or by an internal signal generated by another task or module. The latter may complicate a straightforward feasibility analysis and how to model this behaviour is still under discussion. An advantage of using a preemptive EDF oriented protocol is that EDF has the best utilisation U, defined as $U = \sum_{i=0}^{n} C_i/T_i$, among all other real time schedulers. A possible disadvantage of EDF(I) is it's behaviour under overload, that is when tasks exceed their runtime budget C_i. Under these circumstances it is hard to predict which task will miss a deadline. If this would be a real problem, DCOS could be provided without much effort with a kernel based on Deadline Monotonic scheduling with Deadline Inheritance (DMI), very little effort for the scheduler/dispatcher, however with a partly different feasibility analysis.

4.3 Context Switching

Context switching is a demanding mechanism in processing power, as well as in memory usage. It can be complicated to keep track of the context. Fortunately DCOS inherits the advantages of a single processing stack from the Stack Resource protocol [1] where all tasks share a single stack on which also contexts are saved. If a task is preempted the new context is created just on top of the stack. Restoring a context only occurs when the running task exits after which a preempted task and its context is then refound on top of the stack.

4.4 Task states and queues

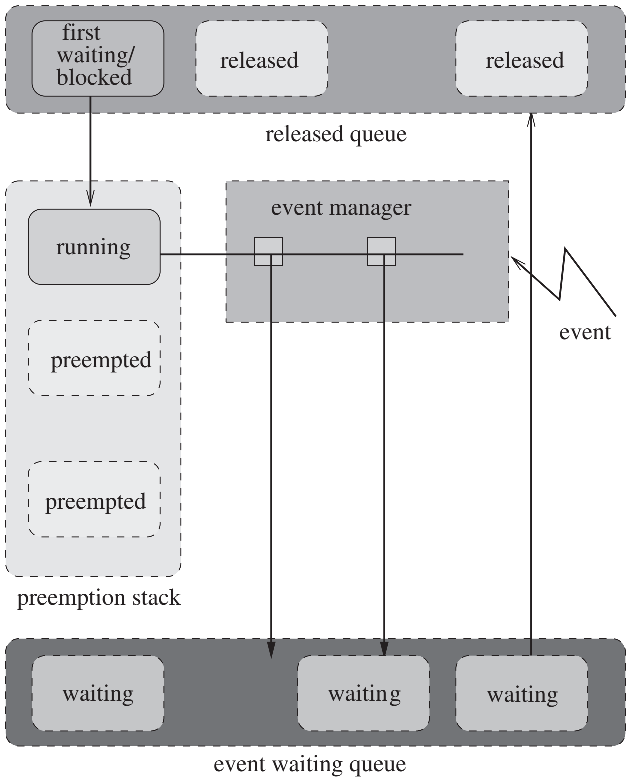

Figure 1. DCOS kernel architecture

The kernel architecture of DCOS tasks is explained with Figure 1. It shows two queues, the waiting queue and the released queue and a stack as well as the transitions of tasks. Every task can reside in one of the following states: waiting, released, running or preempted. Waiting tasks reside in the waiting queue, released tasks reside in the released queue while a preempted task resides on the stack. A running task is associated with the top of the stack. An admitted task is put in the *waiting* queue, in which it waits until an event – either from a periodic or aperiodic interrupt or from a signal – occurs, upon which the target task is transferred to the released queue. Released tasks are ordered to their priority, which is inverse proportional with the absolute deadline. The scheduler will assign the processor to the task with the highest priority. If the head of the released queue has a higher priority than the running task, and if there is no resource conflict, the running task will be preempted in favour of the new high priority task. This causes the running task to become preempted while the new task becomes running. When a running task finishes, it will return to the waiting queue. The scheduler will successively compare the preempted task on top of the stack with the head of the released queue and starts running the task with the highest priority of both. A running task may send signals which in turn may invoke the transfer of waiting tasks to the released queue. In DCOS this may proceed by using DTs, which if symbolicly shown in the figure: on writing a DT its subscribers are signalled via the event manager.

4.5 Resource Synchronization

Resources are elements in your application that can be used by different tasks. A resource can be a variable, a data structure, or a hardware device like a serial port or an LCD. In order to preserve the integrity of a resource in a multitasking system, it must be prevented that two tasks sharing the same resource, have access to it at the same time. A *mutual exclusion* mechanism, original from Dijkstra [3] avoids this concurrent use of shared resources. Mutual exclusion in the DCOS kernel is obtained through the scheduler. First we describe how the use of resources are specified and them we explain how this information is used by the scheduler.

Typically, the access to these resources are subject to read/write restrictions. Each resource is identified by a unique name in the system and every task specification must identify whether it only reads or (also) writes a resource. For convenience we denote read resources by small letters and write resources by capitals. A resource list is organised by *critical sections* in order to group the resources that are used by τ_i simultaneously. For every section it is specified how long a task may hold this section. For instance $\mathcal{R}_i = [t_1(A, b),\ t_2(a, C)]$ specifies that first $(A\ b)$ is hold during a time t_1, where A is written and b is read. Successively the resources are returned to the system after which the following section may be used during a time t_2 where a may be read and C may be written. The value of $\mathcal{R}_i$ is substituted in equation 1.

For the set of tasks that will become active in the system a feasibility analysis will be executed. This is done on basis of deadline inheritance. A resource inherits the lowest relative deadline of any task that needs to use that resource exclusively. Every resource inherits two deadlines, a read deadline and a write deadline. So a read resource, say a, only inherits the lowest deadlines from writers of that resource, while a write resource A inherits the smallest deadline interval from all its readers and writers. If we denote the inherited values of a and A by D_a and D_A then be can conclude that $D_A \leq D_a$.

A critical section asserts the lowest inherited value of all of its resources. A task asserts an inherited deadline interval from the section it executes. If no section is executed a tasks asserts as inherited deadline its original deadline D_i. If we denote the current task's inherited deadline by Δ then a running task τ_r can only be preempted by the head of the released queue τ_h iff:

$$(d_h < d_r) \wedge (D_h < \Delta) \qquad (2)$$

Condition 2 is called the scheduling condition. It provides synchronisation of shared resources. It can be guaranteed that there cannot be two task on the stack that share mutual exclusive resources. However it unavoidably introduces blocking of tasks with a higher priority than the running task if the right-hand of equation 2 is not met. It can however be proved that the higher priority task only can experience blocking once by one task.

5 Results

The DCOS kernel prototype is implemented on a Texas Instruments MSP430 microcontroller running at 4.6MHz. The controller has 2048 bytes of RAM, and 60KB of program flash memory.

Scheduler latency – For the performance metric of the scheduler, we have measured its latency using task-sets of different sizes, ranging from 1 to 16 tasks. Latency is the maximum computation time of the scheduler, and is the time between the activation of the scheduler and the moment the CPU continues, or starts executing a task. The measured latency ranges from 80μs for the smallest task-set, to 110μs for the largest, which is approximately less than double the latency incurred with cooperative scheduling. Based on the measured latency, the maximum number of task switches per second ranges from 9000 to 12500.

Memory usage – For the basic kernel implementation, consisting of the scheduler, data manager, dynamic memory allocator, and the minimum required *Hardware Abstraction Layer* (HAL), the kernel uses 3800 bytes of program flash memory.

The absolute minimum RAM usage of the kernel is 32 bytes and 26 bytes of possible stack usage. For each task an additional 10 bytes of heap space is needed.

6 Conclusion

DCOS has been successfully implemented on a small micro-controller platform to be used for generic support of embedded systems. We have shown that, in the light of the limited amount of energy, hardware resources, it is yet possible to provide a generic operating system – with dynamic memory allocation, management for secondary storage and support for peripheral devices – that allows for a data centric architecture with dynamic loadable modules to be executed while meeting real-time requirements.

References

[1] T. P. Baker. A stack-based resource allocation policy for realtime. In *Proceedings: Real-Time Systems Symposium*, pages 191–200. IEEE Computer Society Press, 1990.

[2] M. Boasson and E. de Jong. Software architecture for large embedded systems. `http://homepages.cwi.nl/~marcello/SAPapers/BJ97.html`.

[3] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Comm. ACM*, 8(9):569, 1965.

[4] E. W. Dijkstra. *Cooperating sequential processes*, pages 43–112. Academic Press, 1968.

[5] S. Dulman, L. van Hoesel, P. Havinga, and P. Jansen. Data centric architecture for wireless sensor networks. In *Proceedings of the ProRISC Workshop*, November 2003.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *ASPLOS 2000*, Nov 2000.

[7] P. G. Jansen and R. Laan. The stack resource protocol based on Real-Time transactions. *IEE Proc.-Software*, 146(2):112–119, Apr 1999.

[8] P. G. Jansen, S. J. Mullender, P. J. M. Havinga, and J. Scholten. Lightweight EDF scheduling with deadline inheritance. Technical report TR-CTIT-03-23, Centre for Telematics and Information Technology, Univ. of Twente, The Netherlands, May 2003. `http://www.ub.utwente.nl/webdocs/ctit/1/000000c6.pdf`.

[9] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of IEEE Real-Time System Symposium*, pages 89–99, Dec 1992.

[10] A. E. Kalman. *Salvo User Manual*. Pumpkin, Inc, 2003.

[11] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proc. Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, pages 41–45, Berlin, Germany, Jan. 2004.

[12] 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.

[13] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67. ACM Press, 2003.

[14] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sep 1990.

[15] L. Sha, R. Rajkumar, and S. Sathaye. Generalised rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1):68–82, Jan 1994.