AmbientRT - real time system software support for data centric sensor networks

#T.J. Hofmeijer†, S.O. Dulman†, P.G. Jansen‡, P.J.M. Havinga ‡
† Ambient Systems; ‡ University of Twente, department of Computer Science, the Netherlands
{hofmeijer,dulman}@ambient-systems.net, {jansen, havinga}@cs.utwente.nl

Abstract

We present the architecture and design of a real time operating system for mobile wireless sensor networks. AmbientRT is being developed for environments with very limited resources in order to relieve the burden of the developer and to efficiently use the resources of the node. This paper presents the main concepts used and trade-offs involved in the system. Initial results show that with the current hardware available for sensor networks, the real time concept is feasible.

For real-time scheduling we have designed EDFI. EDFI is a lightweight real-time scheduling protocol that combines EDF with deadline inheritance over shared resources. EDFI is precise with task admission control, very efficient with scheduling and dispatching, and straightforward in feasibility analysis.

1. INTRODUCTION

The vision of ubiquitous computing requires the development of devices and technologies, which can be pervasive without being intrusive. The basic components of such a smart environment will be small nodes with sensing and wireless communications capabilities, able to organize flexibly into a network for data collection and delivery. Building such a network presents very significant challenges, especially at the architectural and protocol/software level. Major steps forward are required in the field of communications protocol, data processing, and application support. Although sensor nodes will be equipped with a power supply (battery) and embedded processor that makes them autonomous and self-aware, their functionality and capabilities are still very limited.

One key issue that brings wireless sensor networks (WSN) one step closer to reality is the system software running inside each sensor node. Ideally, this software component should allow the user to write his/her own application directly, without spending additional time on managing the low level hardware or the basic functionality of the node. The most convenient framework would be an operating system like software that offers at least real time scheduling mechanisms, memory management and resource management. Additionally, drivers for the existing hardware, as well as design templates for the software components are desirable.

This document describes the design and operation of AmbientRT. The objective of designing AmbientRT was to create a Real Time Operating System that fits inside the limited memory of a sensor node while still supporting low-power modes and causing only minimal overhead. In fact, by supplying higher layer tasks and real time support, the overall efficiency can be improved. Against the intuitive idea that real time scheduling is too expensive in terms of resources, the initial results show that with the current hardware used this is possible and relieves the application programmer to concern about a lot of tiny details.

A. AmbientRT Overview

AmbientRT is a *Real-Time Operating System* for embedded devices with very limited memory, processing, and energy resources. Despite these limitations, AmbientRT has very powerful features like, real-time scheduling, dynamic memory allocation, online reconfigurability, and support for a data driven architecture.

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

Furthermore, with the module support in AmbientRT, applications can be modular in binary format, as well as in code. Firmware can be upgraded by replacing only certain parts, instead of the complete binary. This simplifies the upgrade process, and it conserves precious energy.

B. The unique aspects of AmbientRT

In what way is AmbientRT different to other systems? To answer that question the following list gives the unique aspects of AmbientRT, not seen in comparable systems:

Real-time preemptive EDF scheduling – AmbientRT uses real-time preemptive scheduling [2] on hardware considered having too limited resources to do so. In order to simplify scheduling, other systems use methods as cooperative scheduling [11], where the real-time behaviour is mostly the responsibility of the programmer, or event driven operation like TinyOS [5], 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. Furthermore, with AmbientRT, priorities do not have to be assigned to tasks but are dynamically defined based on the timing properties and the moment in time. This makes the system have a better processor utilization compared to systems that use the aforementioned methods.

Data centric architecture – AmbientRT supports the data centric architecture [4][12]. 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 application compilation only. Supporting the data centric architecture also implies that no added functionality for *Inter Process Communication* (IPC) is needed.

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 can 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 programmer must provide these mechanisms. The latter is a complex task, liable to errors that can cause dead-lock.

2. Related work

A. Operating systems

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 developed at UC Berkeley [5]. 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. Some of its characteristics are: small size of RAM needed, rapid context switch and power aware operation. The task scheduler is very simple because at any time only one task can be run (this allows the usage of only one stack, common for all tasks). The main drawbacks are that it employs cooperative scheduling and has no real time support.

One of the alternatives to TinyOS is the operating system called Salvo [11]. 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 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.

These two examples are not the only operating systems suited for sensor networks (see for example also CMX and others alike). Still they require less resources than their ”bigger brothers” such as: Redhat eCos, Palm OS or Microsoft Windows CE.

B. Real time scheduling

The real time scheduling techniques presented in this paper are based on the combination of EDF and inheritance techniques as presented in [8] for use of shared resources during the complete run-time in tasks and in [6] in which the use of shared resources is organized within nested NCSs. 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.

The foundation for these techniques are based on EDF [13] with the use of shared resources [10] and the inheritance techniques are based on the work of Baker [1] and Sha [16].

The scheduling algorithm is similar to Baker's Stack Resource (SR) protocol [1]. As in SR we use Nested Critical Sections (NCSs), originally introduced by Dijkstra [3] and profitably used by Sha in [16] 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. [15]. 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 [10]. 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.

3. The AmbientRT Kernel

The following gives a short overview of the functionality available in the AmbientRT kernel.

A. Real-Time Scheduler

AmbientRT has an RT-Transactions EDFI scheduler [9][8]. Our technique is based on preemptive Earliest Deadline First (EDF) in a context where shared resources can be used under mutual exclusion. This, in general, complicates scheduling, resource synchronization and switching and confronts the application programmer with a rather complicated environment. We will master this complexity by combining EDF with Deadline Inheritance (DI) and call the resulting scheduling method EDFI.

EDFI is a lightweight preemptive hard real-time scheduling algorithm proved to be deadlock free. Mutual exclusion of shared resources is enforced by the scheduler itself. Through

analysis of a given task-set a guarantee on meeting the real-time constraints for each task can be given.

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.

B. Tools and implementation

We have developed a testing tool for feasibility analysis and offer a graphical web interface [7] for off-line feasibility analysis. The tool can also handle other protocols, like Rate Monotonic with Inheritance (RMI), Deadline Monotonic with Inheritance (DMI) and the Stack Resource protocol [1]. These protocols are beyond the scope of this paper.

The schedulability test can be done off-line or, if on-line, in slack-time. Queue insertions timing costs are at the RT budget and must be covered by the runtime budgets C_i of the tasks. They are however no constant-time operations, but fortunately the queues are invariably short and their contribution is mostly negligible. In addition, the scheduler prevents resource contention from causing gratuitous context switches and it is completely deadlock free. Finally, the same scheduler can trivially be used for preemptive or non-preemptive EDF scheduling.

C. 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, functionality is divided in data producing and consuming components called *Data Centric Entities* (DCE). Data is a generalization of memory objects and *events*, where an event is for example the occurrence of a hardware interrupt. Each distinct memory object or event is called a *Data Type* (DT). The DCEs are used in a *publish/-subscribe* system that allows them to react to DTs produced by others. New configurations can be achieved by altering the set of active DCEs and modifying their subscriptions.

D. Dynamic Loadable Modules

Modules are blocks of software that are not part of the operating system itself. The kernel can load and run modules dynamically. Because of the data centric architecture, these modules can be inserted in the configuration during runtime. This way, AmbientRT is able to support reconfiguration based on functionality becoming available even after device deployment.

A *Dynamic Loadable Module* (DLM) is a task compiled separately from the kernel code. A DLM can be loaded and executed anywhere in program memory, which makes it the building block for creating new configurations online. With this module support in the operating system, modifications to an application can be done more efficient. Instead of updating the complete application (such as described in [14]) 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 heterogenous in the software point of view, resulting in less occupied memory space and better dedicated operation possibilities.

A ready DLM is transferred to the target hardware through for example the radio or the serial port where on arrival it is stored in the secondary storage. For the communication a packet protocol is provided. This protocol divides the DLM into small packets which are uploaded individually so that the node can store it temporary in RAM before writing it to the secondary storage. When the node successfully received a complete packet, the sender will send the next packet. The protocol can be used for any type of binary that has to be uploaded to the secondary storage. A low complexity file system is used for creating, and keeping track of files representing the received binaries in the secondary storage.

E. Dynamic Memory Allocation

AmbientRT is able to dynamically reconfigure itself by altering its task - and data sets. So a task - or data set can grow or shrink in size. To prevent the need of reserving the maximum allowed size for these sets, and therefore wasting precious memory, AmbientRT supports dynamic memory allocation to reserve and free memory space of arbitrary size in a dedicated heap area.

4. REAL-TIME OPERATION

The AmbientRT kernel is a real-time kernel. This kernel is based on fundamentals laid down in the Stack Resource Protocol [8]. The following will describe the real-time concepts and operation of AmbientRT.

A. Real-time tasks

A task, or process, is a subprogram of an application that performs an action. A real-time task has added time constraints, to indicate that the action should be completed at a certain moment in time.

A task in AmbientRT is a real-time task. It is defined by a set of properties indicating its time behaviour, time constraints, and resource usage. These properties, which must be provided by the application designer, are listed below:

- *Deadline* is the time relative to the moment the task is selected for scheduling, before which the task should finish. The deadline of a task i is indicated by D_i.
- *Period* is the time between every activation of a task, and is used in the task-set analysis. The period is notated as T_i for a task i. If the period is unknown or the task is aperiodic, the minimal possible time between two consecutive executions of the task should be used.

- *CPU Cost* is the worst-case computation time of a task i, notated as C_i. This value is also used in the task-set analysis.
- *Resource Usage* is a list of resources the task uses. Resources are identified by a unique name in the system. For every resource in the list is specified if the task needs read or write access to it. The Resource Usage of a task i is notated as P_i.

Once a task is active in the system, the following dynamic properties are assigned to it:

- *Release time* is the moment a task is selected for scheduling. The release time of a task i is notated as: r_i.
- *Absolute deadline* is the exact time of the deadline of a task. It is calculated by adding the deadline to the release time ($r_i + D_i$). For a task i the absolute deadline is notated as d_i.

B. Real-time Multitasking

The real-time scheduler in AmbientRT uses dynamic priorities. This means that the priority of a task relative to the priorities of other tasks changes over time. The scheduler uses the absolute deadline as the priority of a task. The task with the earliest absolute deadline has the highest priority.

Figure 1 shows an example of two tasks being scheduled. Task B is running since r_B when an event causes task A to be selected for scheduling. The release time of A is r_A. Because the absolute deadline of task A (d_A) is earlier than the one of task B (d_B), task A is of higher priority and is assigned to the processor. After task A finishes, task B completes its operation.

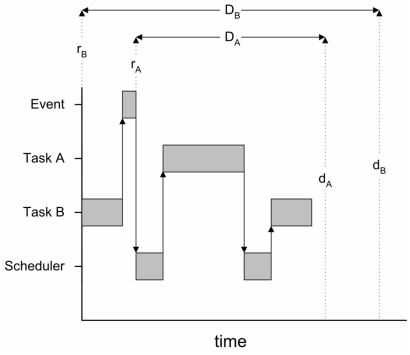

Fig. 1: Scheduling two real-time tasks I

The impact of using dynamic priorities can be seen in Figure 2. In this case the duration of task B is extended and the event that causes task A to be scheduled comes later. Although task A has a smaller deadline ($D_A < D_B$), it has a later absolute deadline ($d_A > d_B$), and is therefore of lower priority than task B. If static priorities should have been used here, task B would have been suspended by task A and wouldn't have completed on time. Using the absolute deadline as the priority in general, leads to a better utilization of the processor than when fixed priorities are used.

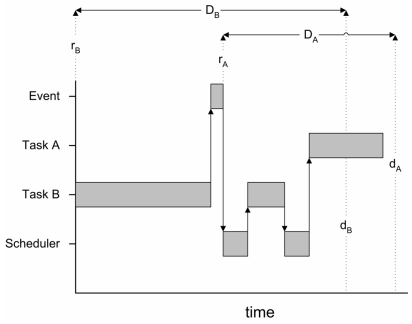

Fig. 2: Scheduling two real-time tasks II

C. Context Switching

The *context* of a task is its processor state, usually the contents of the registers and the stack. When a running task is suspended and a different task will start or continue execution, the kernel performs a *context switch*, which means that the context of the running task is saved, and the context of the new task is created or restored. Context switching is a demanding mechanism in processing power, as well as in memory usage.

The advantage of the AmbientRT kernel is that the tasks in the system all share a single stack. Because of this the context of a task doesn't have to be saved somewhere else but can be left on the stack itself. If a task is preempted the new context will be created just on top of the old one. Restoring a context only occurs when the running task exits and a preempted task continues. The task that will continue is always the task that has its context directly below the running task, and therefore restoring a context is nothing more than removing the context of the running task.

D. Task states

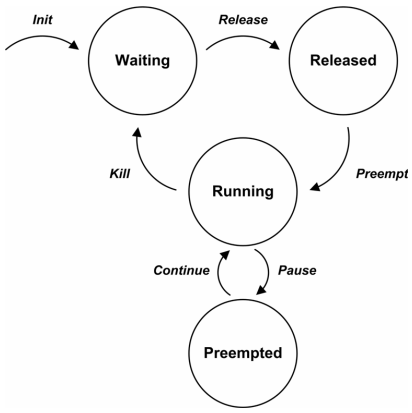

Fig. 3: Scheduler task state transition diagram

In AmbientRT a task can be in several states. Figure 3 illustrates the state transition diagram of a task. Every task is created in the *waiting* state, in which the task will wait until an event causes it to be transferred to the *released* state. A task in the released state must run on the processor as soon as possible. The scheduler assigns the processor to the task in

the released state with the highest priority. This task is then transferred to the *running* state. When a task is in the running state and a new task is transferred to the released state, the priority of the new task is compared with that of the running task. If the new task has a higher priority, then the running task will be preempted, or paused. The old task is transferred to the *preempted* state, and the new task will transfer to the running state. When a task in the running state finishes, it will be killed and moved back to the waiting state. The scheduler will then compare the highest priority preempted task to the highest priority released task. If the preempted task is of higher priority, it is moved back to the running state where it will continue its operation. If not, the released task will be moved to the running state. At all times, on a single processor platform, only one task can be in the running state.

E. Resource Synchronization

Resources are elements in an application that can be used by different tasks. A resource can be a variable or 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 avoids this concurrent use of un-shareable resources.

Mutual exclusion in the AmbientRT kernel is obtained through the scheduler. It provides automatic synchronization of shared resources. The scheduler compares on initialization the resource usage lists of every task and generates per task, on basis of the relative deadlines, a threshold value. Comparing this threshold value of a task to the deadline of another indicates whether they share a resource. When the scheduler determines which task is allowed to run it will not only compare the dynamic priorities of the tasks, but also the threshold of the running task to the deadline of the candidate task. If they share a resource, the scheduler will first let the running task finish even if the candidate task has an earlier deadline. In this last situation the candidate task is *blocked* by the running task, and the added delay because of this is called the *blocking time*.

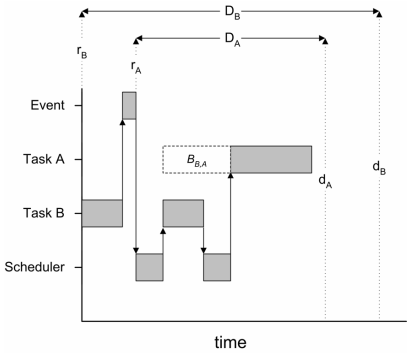

Fig. 4: Scheduling two tasks that share a resource

This mechanism is illustrated by Figure 4. In this case task A and B share a resource. This time, when the event occurs, the scheduler let task B finish first. After task B is done, the scheduler runs task A. The blocking time is indicated by $B_{B,A}$.

5. RESULTS

The AmbientRT 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

This paper gave an introduction to AmbientRT – system support for very small embbeded systems, such as wireless sensor networks.

The key characteristics of AmbientRT are 1) its real-time EDF scheduling, 2) its automatic mutual exclusion, and 3) its support for data centric architectures.

AmbientRT has been successfully implemented on a small micro-controller platform. A major advantage of our approach is that little work is needed at application level. Synchronization obligations of tasks, due to resource usage, are almost completely shifted to the system level. Also the feasibility analysis and admission control is done at system level and it can be carried out off-line or in idle time so that they do not burden the real-time budget. A new task can be admitted after the analysis has been carried out successfully.

The real-time scheduler will be available under GNU Public License.

REFERENCES

[1] T. P. Baker. Stack-based scheduling of real-time processes. *The journal of real-time systems*, 3(1):67–99, 1991.

[2] S. Baruah, A. Mok, and L. Rosier. Preemptive scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the Real-Time Systems Symposium*, pages 182–190, Dec 1990.

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

[4] 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.

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

[6] P. Jansen, S. Mullender, and P. J. Havinga. Lightweight edf scheduling with deadline inheritance. In *Submitted for publication to Proceedings 16th Euromicro Conferxence on Real-Time Systems*, Calabrie, Sicily, Jun 2004. IEEE Computer Society Press.

[7] P. G. Jansen and F. Hanssen. Clockwork – a Real-Time feasibility analysis tool (software), Sep 2001.

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

[9] 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`.

[10] 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.

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

[12] 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)*, Jan. 2004.

[13] 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.

[14] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. Technical report, Faculty of Information Technology and Systems, Delft University of Technology, The Netherlands, October 2003.

[15] 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.

[16] 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.